Deep Learning

Class II: Convolutional Neural Networks

Hédi Hadiji

Université Paris-Saclay - CentraleSupelec

hedi.hadiji@l2s.centralesupelec.fr

October, 2025

Table of Contents

Last time

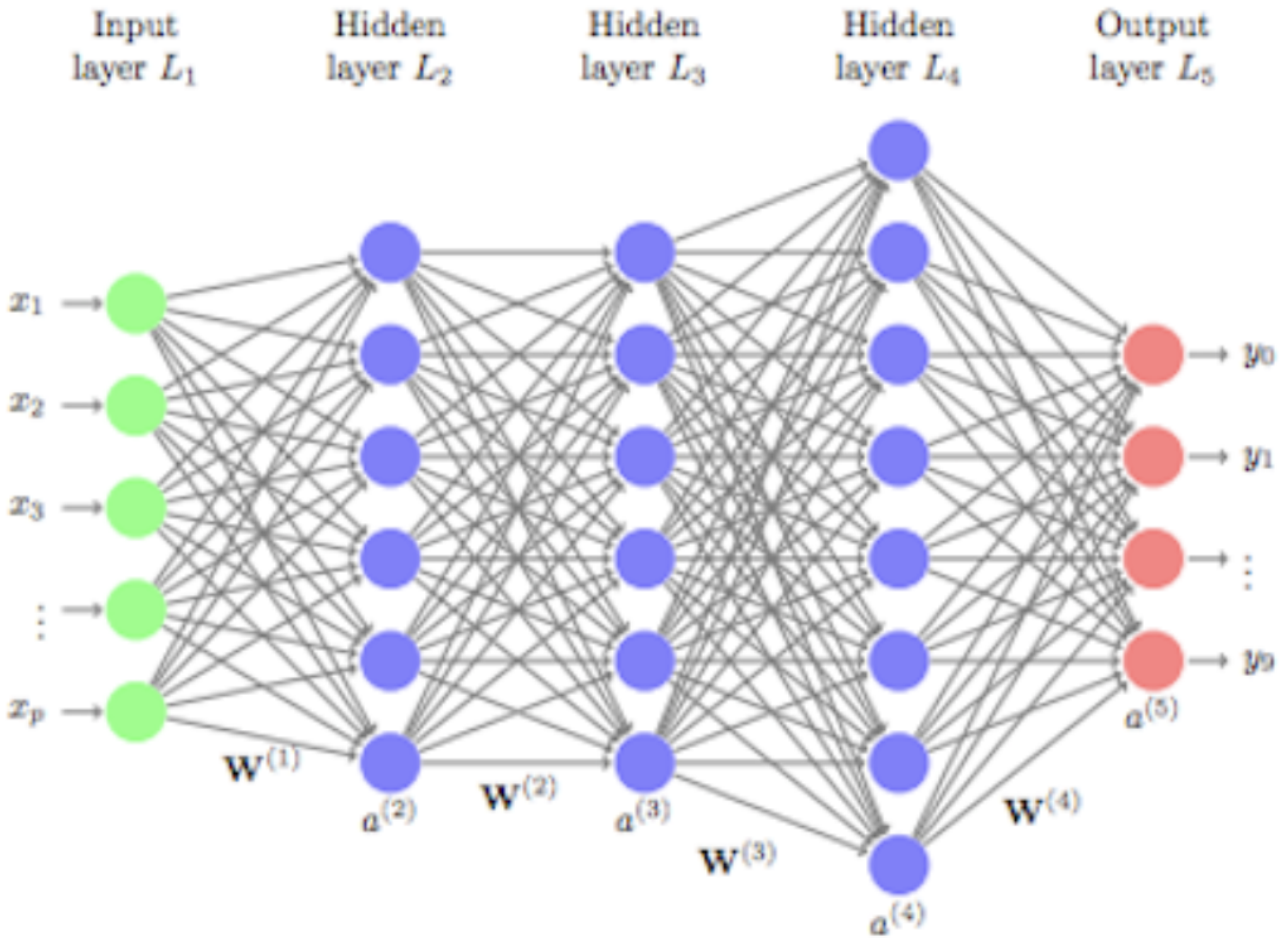
Last time:

- Supervised learning
- Neural nets
- Multi-Layer Perceptron + Backpropagation

Today: Convolutional Neural Networks

Table of Contents

Recall: MLP


Input layer L_1
Hidden layer L_2
Hidden layer L_3
Hidden layer L_4
Output layer L_5
x_1
x_2
x_3
x_p
y_0
y_1
y_9
$\mathbf{W}^{(1)}$
$\mathbf{W}^{(2)}$
$\mathbf{W}^{(3)}$
$\mathbf{W}^{(4)}$
$a^{(2)}$
$a^{(3)}$
$a^{(4)}$
$a^{(5)}$

Softmax

- Given logits $z \in \mathbb{R}^K$ the softmax converts them to a probability vector

$$\text{softmax}(z)_j = \frac{\exp(z_j)}{\sum_{i=1}^{K} \exp(z_i)}.$$

- Common loss: cross-entropy with one-hot label y,

$$\mathcal{L}(z, y) = -\sum_j y_j \log \text{softmax}(z)_j,$$

  (libraries implement it directly as a function of logits)
- Gradient (softmax + cross-entropy, compact form):

$$\frac{\partial \mathcal{L}}{\partial z} = p - y, \qquad p = \text{softmax}(z),$$

  and Jacobian entries $\partial p_i / \partial z_j = p_i(\delta_{ij} - p_j)$.

Recall: Backprop

- Forward: compute result and intermediate activations
- Backward: propagate gradients going from deep to shallow layers.

Backpropagation Algorithm

Algorithm 1 Backpropagation for an L-layer network

Given: input x, target y, parameters $\{W_\ell, b_\ell\}_{\ell=1}^{L}$
Forward pass:
for $\ell = 1 \to L$ **do**
    $z_\ell \leftarrow W_\ell a_{\ell-1} + b_\ell$
    $a_\ell \leftarrow \sigma_\ell(z_\ell)$
Compute loss $J = \mathcal{L}(a_L, y)$

Backward pass:
$\delta_L \leftarrow \nabla_{a_L} \mathcal{L}(a_L, y) \odot \sigma_L'(z_L)$
for $\ell = L - 1$ **down to** 1 **do**
    $\delta_\ell \leftarrow (W_{\ell+1}^\top \delta_{\ell+1}) \odot \sigma_\ell'(z_\ell)$

Gradients:
for $\ell = 1 \to L$ **do**
    $\nabla_{W_\ell} J \leftarrow \delta_\ell a_{\ell-1}^\top$
    $\nabla_{b_\ell} J \leftarrow \delta_\ell$

Complexity of computations

- Consider an MLP with L layers. Layer l maps $n_{l-1} \to n_l$ (weight matrix W_l of size $n_l \times n_{l-1}$).
- Forward pass cost (per input):

$$T_{\text{fwd}} = \Theta\Big(\sum_{l=1}^{L} n_{l-1} n_l\Big)$$

  (matrix-vector products + activation costs).
- Backward pass cost (per input): of the same order — computing layer deltas and weight gradients involves similar matrix products: often quoted as roughly 2-3 times the forward cost.
- Memory: need to store activations for each layer during forward for use in backprop:

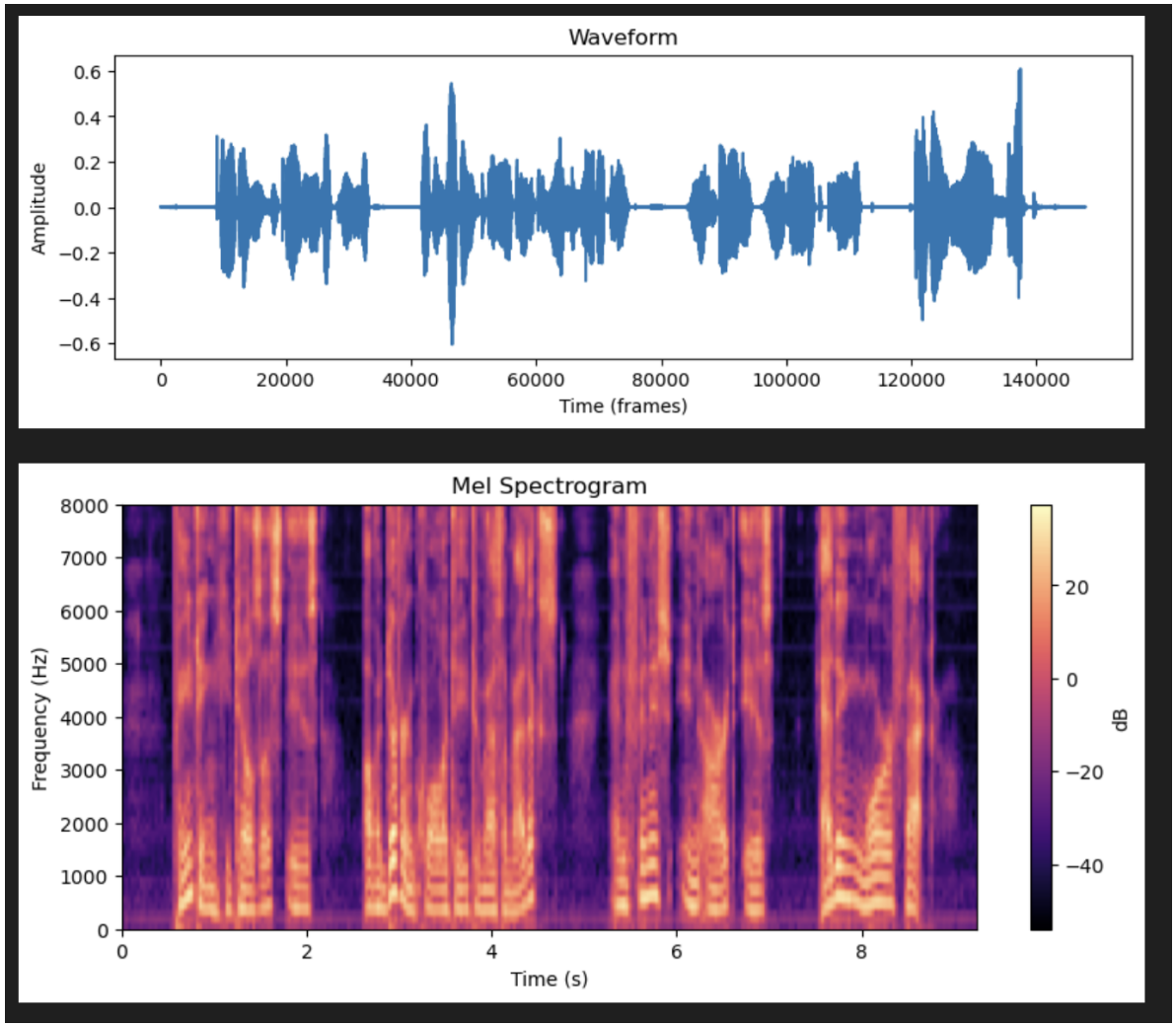
$$M_{\text{act}} = \Theta\Big(\sum_{l=0}^{L} n_l\Big)$$

  plus parameter storage $\Theta(\sum_l n_{l-1} n_l)$.

Table of Contents

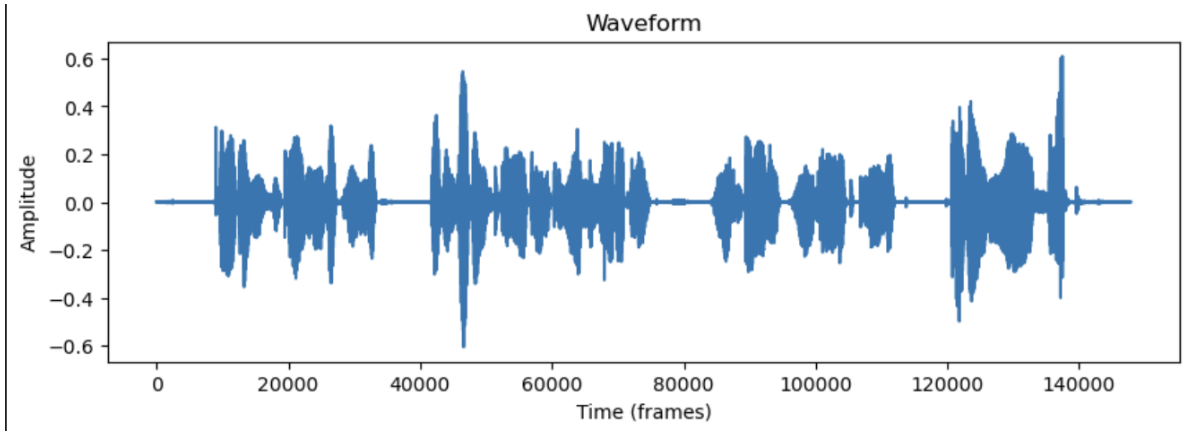
Question: What if your task has some invariance?

Example: Speech signal



Task: predict whether the word hello is pronounced

What happens when you input this in an MLP?

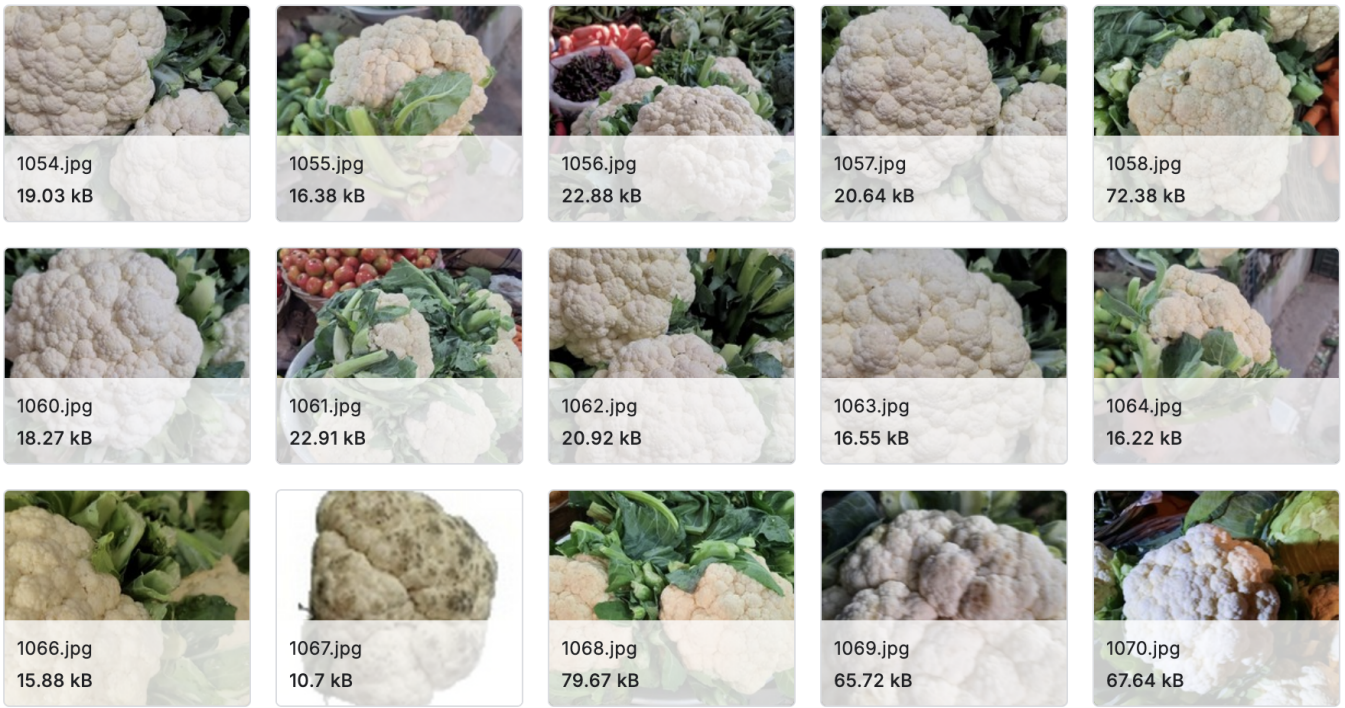


Input Dimension: 150 000, can be downsampled to e.g. 15 000. Number of parameters is at least 15 000 $\times$ hidden dimension, much more if multiple layers.
Information for task is located in time.

- Not data efficient: network will need to see interesting data at every location until all weights get updated meaningfully
- A better way would be to find a way to use the shift invariance of the task in the model architecture.

Image processing

Same with images in 2D.



We build our understanding of what is in a image by through local features that we combine together.
Idea: instead of fully-connected layers, apply a local operation.

Table of Contents

Formal Definition of Convolution

For an input signal $x = (x_t)_{t \in [T]}$ and kernel $(w_k)_{k \in [K]}$:

$$(x * w)_t = \sum_{k=0}^{K-1} w_k x_{t+k}$$

- K is called the kernel size
- Convolution is a product computed at every shift
- Called 'cross-correlation' in signal processing (not true convolution: convolution would flip the kernel)

Example: Simple Convolution

Input: $[1, 2, 3, 4, 5]$ Kernel: $[1, 0, -1]$

$$y[0] = 1 \cdot 1 + 2 \cdot 0 + 3 \cdot (-1) = -2$$
$$y[1] = 2 \cdot 1 + 3 \cdot 0 + 4 \cdot (-1) = -2$$
$$y[2] = 3 \cdot 1 + 4 \cdot 0 + 5 \cdot (-1) = -2$$

Output: $[-2, -2, -2]$

Output is:

- Positive if sequence is incresing
- Negative if sequence is decreasing

$\Rightarrow$ This kernel detects decreasing transitions (like a 1D edge detector).

Visualizing the Operation

- Multiply elementwise $\rightarrow$ sum $\rightarrow$ slide one step $\rightarrow$ repeat.

See **this demo**

Stride and Padding

Extra-parameters at implementation

- **Stride** = how many steps we slide the kernel each time.
  - Stride 1 : dense output
  - Stride 2 : downsampled output

  Stride reduces the size of the output, at the cost of losing information
- **Padding** = add zeros around edges to control output length.
  - ”valid” = no padding
  - ”same” = output length = input length

  Padding affects how we look at the borders of the image.

Multi-Channel Conv1D

Instead of sequences of numbers, we can look at sequences of vectors: each coordinate is a channel.
We can apply a filter that maps every time index vector to another vector instead.
Input can have multiple channels (e.g. stereo sound, multiple sensors, ...)
Each x_t is itself a vector in $\mathbb{R}^{N_{\text{channels}}}$.
Each time-component of the kernel is itself a vector, so full kernel is a matrix $w \in \mathbb{R}^{K, N_{\text{channels}}}$

$$(x * w)_t = \sum_{k=0}^{K-1} w_k \cdot x_{t+k}$$

- Each kernel spans all input channels.
- Output is still 1-dimensional

A kernel captures interactions locally in time, but cross-channel.
Finally, we can also have multiple output channels. Then the kernel w is represented as tensor.

Why Use Conv1D?

- Captures local temporal or spatial dependencies.
- Parameter efficient: same kernel reused across positions.
- Translation invariant: pattern recognized anywhere in input.
- Common in:
  - Audio / waveform processing
  - Time series forecasting
  - NLP on embeddings
  - Reinforcement learning (temporal signals)

Strength of deep learning: principled way of incorporating knowledge about the task without hardcoding features.

Minimal NumPy Implementation

```
import numpy as np

def conv1d(x, w, stride=1):
        n, k = len(x), len(w)
        out_len = (n - k) // stride + 1
        y = np.zeros(out_len)
        for i in range(out_len):
                start = i * stride
                y[i] = np.sum(x[start:start+k] * w)
        return y
```

Table of Contents

Generalization to Conv2D

$$(w * x)_{i,j} = \sum_{m,n} w_{m,n} \cdot x_{i+m,j+n}$$

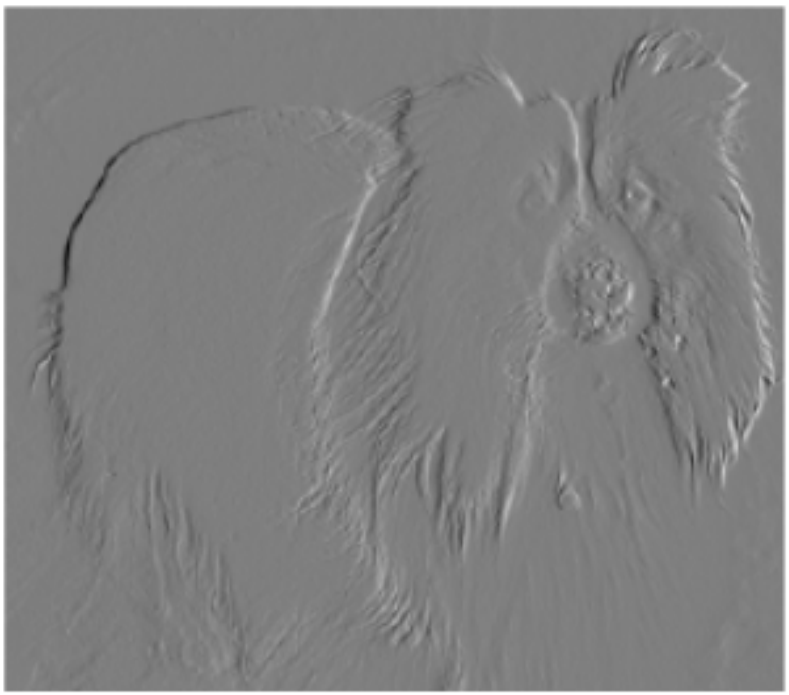
- Same concept, just over two spatial axes (height, width)

Question

What happens if you apply the kernel $[-1, 1]$ to this image:

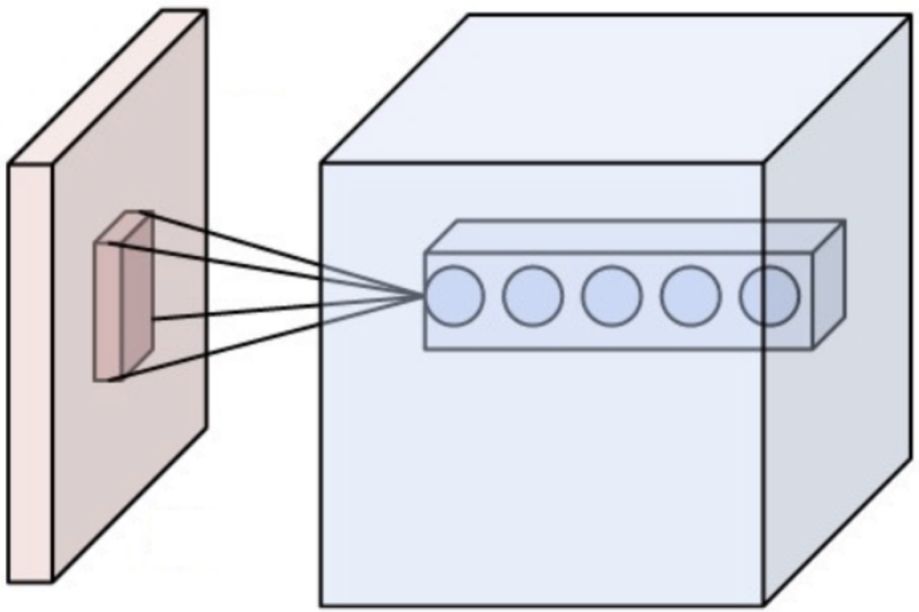
2D Kernels

Applying the kernel $[-1, 1]$



Detect vertical edges.
Source: Goodfellow, Bengio, Courville.

Images as Blocks



With channels, we can see an image as a 3D block.
And convolutions as (linear) mappings between 3D blocks that act only locally on the first two dimensions.

Demo Visualization

See **this animation**

- Convolution ”scans” for features by applying the same local rule everywhere.
- A neural network then attempts to learn the correct filter for the task.

Table of Contents

Convolutions by themselves are not expressive enough

Why?

$$x \mapsto (w * x)_{i,j} = \sum_{m,n} w_{m,n} \cdot x_{i+m,j+n}$$

Second Ingredient: Non-linearity

Add a ReLU.

Third Ingredient: Pooling Layers

Motivation:

- Reduce the spatial size of feature maps (width and height)
- Preserve important features while reducing redundancy
- Make the network more robust to small translations and noise

How it works:

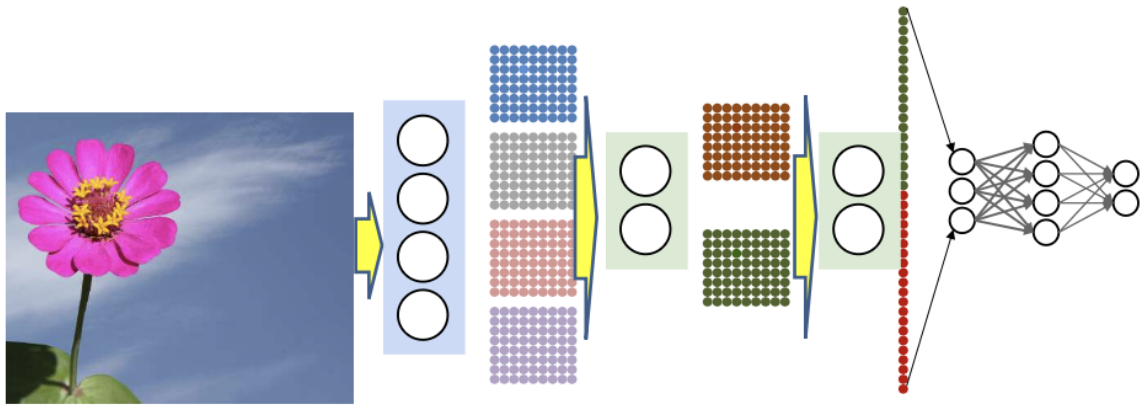
- Divide each feature map into small regions (e.g. 2×2)
- Apply an aggregation function in each region:
  - **Max pooling:** take the maximum value
  - **Average pooling:** take the mean value

Remark:

- Pooling is not so favored anymore in modern archicture besides very light-weight settings: downsampling by using strides is better in practice.

Learning Features and Hierarchical Learning

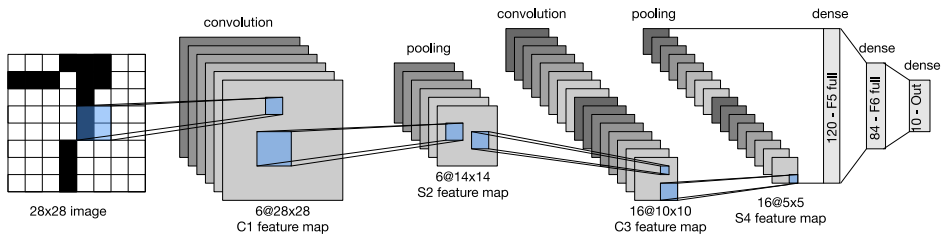
Apply a convolutional neural network:



Flatten output then put in an MLP (aka dense layers, aka fully-connected layers) for a classification task.

Historic(al) architectures

LeNet (1998): Hand-written digit recognition



Alexnet (2012): Breakthrough on image classification

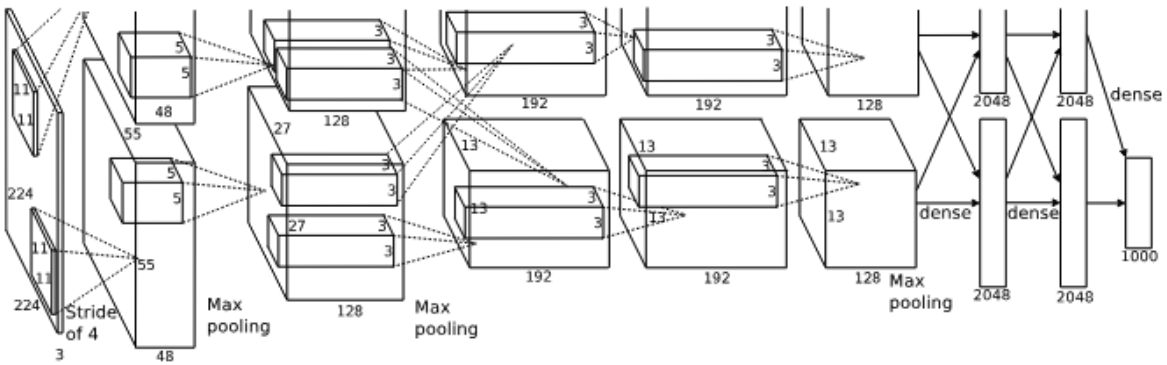


Figure 2: An illustration of the architecture of our CNN, explicitly showing the delineation of responsibilities between the two GPUs. One GPU runs the layer-parts at the top of the figure while the other runs the layer-parts at the bottom. The GPUs communicate only at certain layers. The network's input is 150,528-dimensional, and the number of neurons in the network's remaining layers is given by 253,440–186,624–64,896–64,896–43,264–4096–4096–1000.

Learned features

We can plot the weights of the first layer in a trained network:

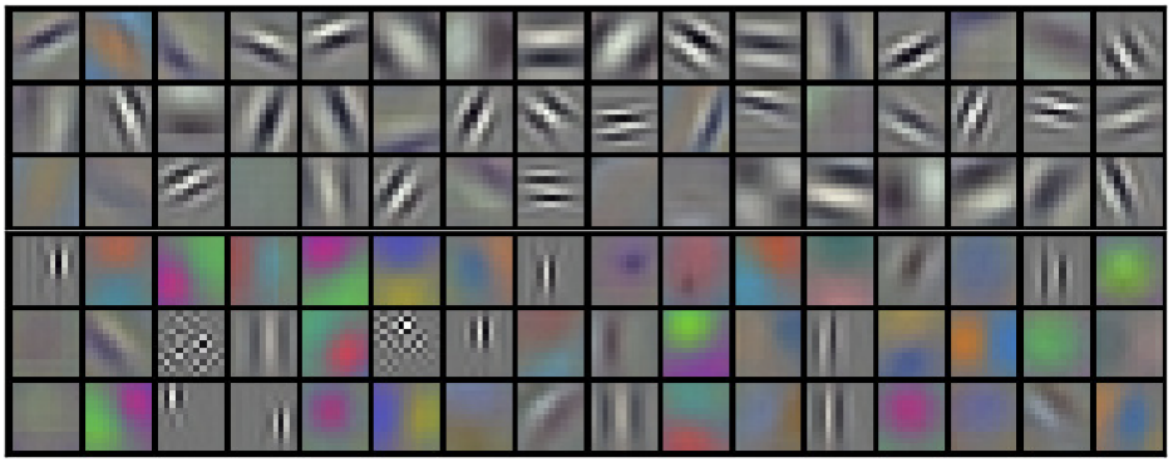


Figure 3: 96 convolutional kernels of size $11 \times 11 \times 3$ learned by the first convolutional layer on the $224 \times 224 \times 3$ input images. The top 48 kernels were learned on GPU 1 while the bottom 48 kernels were learned on GPU 2. See Section 6.1 for details.

In TP

- Basics of pytorch
- Implement backprop