Convolutional Neural Networks

Handout Notes - Generative and Deep Learning (GDL)

Davide Bacciu - University of Pisa

Notation. Images are denoted by bold lowercase tensors such as $\mathbf{x} \in \mathbb{R}^{H \times W \times C}$, where H is height, W is width, and C is the number of channels. A convolution kernel is denoted by $\mathbf{K}$, typically of size $K_h \times K_w \times C_{\text{in}}$, and a bank of kernels produces C_{out} output channels. Feature maps are denoted by $\mathbf{h}$ or $\mathbf{a}$. Stride is denoted by S, padding by P, and dilation by d. A dataset is denoted by $\mathcal{D}$ with size $N = |\mathcal{D}|$. Model parameters are denoted by θ. When needed, activations at layer ℓ are written $\mathbf{h}^{(\ell)}$.

1 Why convolutional networks?

Convolutional neural networks (CNNs) are deep architectures designed for data with a strong spatial or sequential organization, especially images, audio, and other grid-like signals. Their key success comes from introducing the right *inductive bias*: instead of treating every input coordinate as unrelated to every other one, CNNs assume that nearby locations are strongly related and that similar local patterns may appear at different positions.

For images this assumption is highly natural. A cat remains a cat whether it appears near the center or near a corner of the image. Edges, corners, textures, and object parts are local patterns that can recur anywhere. A model should therefore:

- preserve spatial neighborhood relations,
- reuse the same detector at different locations,
- gradually aggregate local information into larger and more abstract patterns.

A fully connected multilayer perceptron is poorly matched to this structure. If we flatten an image into a vector, spatial locality is largely destroyed, the number of parameters becomes very large, and the network must learn from scratch that the same local pattern should be recognized regardless of its position. CNNs address all three issues simultaneously through local connectivity, weight sharing, and hierarchical composition.

2 Historical perspective

The core ideas behind CNNs go back well before the current deep learning era. Biological inspiration came from the study of the visual cortex, especially the distinction between *simple cells*, sensitive to localized features, and *complex cells*, which pool over local variations to gain invariance. These ideas influenced early hierarchical vision systems such as the *Neocognitron*.

The first trainable convolutional architectures appeared in the 1980s, both for sequences and for images. Time-delay neural networks used shared local filters over sequences, while LeCun's early convolutional networks for images established the now-familiar pattern of convolution, nonlinearity, pooling, and dense classification layers. The major breakthrough in large-scale vision came later, when improved optimization, better hardware, more data, and better regularization made deep CNNs practical. Since then, architectures such as AlexNet, VGG, GoogLeNet/Inception, ResNet, and DenseNet have progressively refined the same basic design principles.

3 Images as structured inputs

A grayscale image can be represented as a matrix

$$\mathbf{x} \in \mathbb{R}^{H \times W},$$

while an RGB image is a three-dimensional tensor

$$\mathbf{x} \in \mathbb{R}^{H \times W \times 3}.$$

The third dimension corresponds to channels. More generally, feature maps inside a CNN also have a channel dimension.

If such an image is flattened into a vector of length HWC, a dense layer computes

$$\mathbf{h} = W\mathbf{x} + \mathbf{b},$$

where W is a dense parameter matrix. This representation has two problems. First, the number of parameters scales with the full input size. Second, the learned features are position-specific: a detector for a vertical edge in the top-left corner is represented by different parameters than a detector for the same edge elsewhere. This is contrary to the translational structure of images.

CNNs replace dense matrix multiplication with convolution, thereby enforcing sparse local connectivity and parameter sharing.

4 Discrete convolution for images

For our purposes, the basic operation is discrete 2D convolution (in deep learning, often implemented as cross-correlation; the distinction is not important for learning because the kernel is learned). Let $\mathbf{x}$ be an input image and $\mathbf{K}$ a kernel with finite support. A scalar output at spatial location (i, j) is computed by a weighted sum over a local neighborhood:

$$(\mathbf{x} * \mathbf{K})(i, j) = \sum_m \sum_n \mathbf{x}(i - m, j - n)\, \mathbf{K}(m, n).$$

In most CNN implementations one instead writes

$$(\mathbf{x} \star \mathbf{K})(i, j) = \sum_m \sum_n \mathbf{x}(i + m, j + n)\, \mathbf{K}(m, n),$$

which is cross-correlation rather than strict convolution. Since $\mathbf{K}$ is learned, the network can absorb the flip, so both lead to the same modeling capacity. For simplicity, we will use the term "convolution" for the operation performed in CNNs.

The crucial point is that the *same kernel coefficients* are reused at every spatial location. This is exactly the parameter-sharing mechanism that gives CNNs translation-aware behavior.

Worked example — A local convolution response as a neuron computation

Consider a 3×3 kernel applied to a single-channel image. Let

$$\mathbf{K} = \begin{bmatrix} w_1 & w_2 & w_3 \\ w_4 & w_5 & w_6 \\ w_7 & w_8 & w_9 \end{bmatrix}, \qquad \mathbf{x}_{i,j} = \begin{bmatrix} x_{i-1,j-1} & x_{i-1,j} & x_{i-1,j+1} \\ x_{i,j-1} & x_{i,j} & x_{i,j+1} \\ x_{i+1,j-1} & x_{i+1,j} & x_{i+1,j+1} \end{bmatrix}.$$

Then the pre-activation at position (i, j) is

$$a_{i,j} = \sum_{u=1}^{3} \sum_{v=1}^{3} w_{uv}\, x_{uv} + b = w_1 x_{i-1,j-1} + w_2 x_{i-1,j} + \cdots + w_9 x_{i+1,j+1} + b.$$

This is exactly a neuron computation of the form

$$a_{i,j} = \mathbf{w}^\top \mathbf{x}_{i,j} + b,$$

but performed only on a local receptive field, and with the same $\mathbf{w}$ reused everywhere in the image.

Thus a convolutional unit can be understood as a sparse neuron with shared parameters.

5 Multi-channel convolution

Real images and intermediate feature maps usually have multiple channels. If the input is

$$\mathbf{x} \in \mathbb{R}^{H \times W \times C_{\text{in}}},$$

then one convolution kernel must span all input channels:

$$\mathbf{K} \in \mathbb{R}^{K_h \times K_w \times C_{\text{in}}}.$$

The output at one location is obtained by summing over both spatial offsets and channels:

$$a_{i,j} = \sum_{c=1}^{C_{\text{in}}} \sum_{m=1}^{K_h} \sum_{n=1}^{K_w} \mathbf{x}_{i+m,j+n,c}\, \mathbf{K}_{m,n,c} + b.$$

Therefore one kernel produces one output feature map. To obtain multiple output channels, we use a *bank* of kernels:

$$\mathbf{K}^{(1)}, \ldots, \mathbf{K}^{(C_{\text{out}})}.$$

The result is an output tensor

$$\mathbf{a} \in \mathbb{R}^{H' \times W' \times C_{\text{out}}}.$$

This is one of the most important bookkeeping facts in CNNs: spatial dimensions usually change according to kernel size, stride, and padding, whereas the channel dimension changes according to the number of kernels.

6 Stride and output size

Basic convolution moves the kernel one pixel at a time, which corresponds to stride $S = 1$. More generally, stride is a hyperparameter that controls how far the kernel moves at each step. A larger stride reduces computational cost and downsamples the feature map.

If the input has spatial size $H \times W$, the kernel is $K_h \times K_w$, padding is P_h, P_w, and stride is S_h, S_w, then the output size is

$$H' = \left\lfloor \frac{H - K_h + 2P_h}{S_h} \right\rfloor + 1, \qquad W' = \left\lfloor \frac{W - K_w + 2P_w}{S_w} \right\rfloor + 1.$$

For square kernels, equal padding, and equal stride, this simplifies to

$$H' = \left\lfloor \frac{H - K + 2P}{S} \right\rfloor + 1, \qquad W' = \left\lfloor \frac{W - K + 2P}{S} \right\rfloor + 1.$$

Worked example — Computing the output size of a convolution

Let an input image have size 7×7, let the kernel size be 3×3, and let stride be $S = 2$ with no padding. Then

$$H' = \left\lfloor \frac{7 - 3}{2} \right\rfloor + 1 = \lfloor 2 \rfloor + 1 = 3,$$

and similarly

$$W' = \left\lfloor \frac{7 - 3}{2} \right\rfloor + 1 = 3.$$

So the output size is

$$3 \times 3.$$

If instead stride were $S = 1$, the output would be

$$(7 - 3) + 1 = 5,$$

hence a 5×5 feature map.

Stride is therefore both a computational and representational design choice: it reduces the number of multiplications and acts as a learned form of downsampling.

7 Zero padding

Without padding, convolution shrinks the spatial dimensions because the kernel cannot be centered near the border without going outside the image. To control this shrinkage, CNNs commonly use *zero padding*: rows and columns of zeros are added around the image border.

If padding is P and the kernel size is K, the output-size formula becomes

$$H' = \left\lfloor \frac{H - K + 2P}{S} \right\rfloor + 1, \qquad W' = \left\lfloor \frac{W - K + 2P}{S} \right\rfloor + 1.$$

A common choice for odd kernels and stride 1 is

$$P = \frac{K - 1}{2},$$

which preserves spatial size:

$$H' = H, \qquad W' = W.$$

Padding serves several purposes:

- preserving spatial size across layers,
- allowing border pixels to influence outputs as much as central pixels,
- making architectural design more regular.

8 Nonlinearity after convolution

Convolution is a linear operation. If a network used only convolutions and no nonlinearities, then the composition of layers would still be linear, regardless of depth. To obtain expressive representations, CNNs apply an elementwise nonlinearity after the convolution:

$$\mathbf{h} = \phi(\mathbf{a}),$$

where $\mathbf{a}$ is the pre-activation tensor and ϕ is typically ReLU:

$$\phi(z) = \max(0, z).$$

The resulting pattern

$$\text{convolution} \;\rightarrow\; \text{bias} \;\rightarrow\; \text{nonlinearity}$$

forms the core of a convolutional feature extractor.

ReLU is especially important in deep networks because it reduces gradient-vanishing effects relative to saturating nonlinearities such as sigmoid or tanh. For positive inputs its derivative is 1, which helps gradient flow; for negative inputs it outputs zero, which also encourages sparse activations.

9 Pooling and invariance

Pooling is an operation applied to a feature map to produce a smaller and often more robust representation. The most common choice is *max pooling*. For a local window, max pooling outputs the maximum activation in that window. For example, a 2×2 max-pooling layer with stride 2 maps a 4×4 feature map to a 2×2 feature map.

If the pooling window has size K and stride S (with no padding, which is common), the output size follows the same formula as above:

$$H' = \left\lfloor \frac{H - K}{S} \right\rfloor + 1, \qquad W' = \left\lfloor \frac{W - K}{S} \right\rfloor + 1.$$

Pooling has two effects:

- it reduces spatial resolution,
- it provides a degree of local translation invariance.

If a feature is detected slightly shifted within the pooling window, the pooled response may remain unchanged.

Worked example — Max pooling on a small feature map

Consider the feature map

$$\begin{bmatrix} 1 & 1 & 2 & 4 \\ 5 & 6 & 7 & 8 \\ 3 & 2 & 1 & 0 \\ 1 & 2 & 3 & 4 \end{bmatrix}.$$

Apply 2×2 max pooling with stride 2. The four pooling windows are:

$$\begin{bmatrix} 1 & 1 \\ 5 & 6 \end{bmatrix}, \quad \begin{bmatrix} 2 & 4 \\ 7 & 8 \end{bmatrix}, \quad \begin{bmatrix} 3 & 2 \\ 1 & 2 \end{bmatrix}, \quad \begin{bmatrix} 1 & 0 \\ 3 & 4 \end{bmatrix}.$$

Taking the maximum in each window yields

$$\begin{bmatrix} 6 & 8 \\ 3 & 4 \end{bmatrix}.$$

Thus the output is a 2×2 pooled map.

Max pooling is the most common form, but average pooling, L^2 pooling, and global average pooling are also important in practice.

10 The convolutional layer as a module

In practice, one often treats

$$\text{convolution} \;\to\; \text{nonlinearity} \;\to\; \text{pooling}$$

as a single conceptual module, although modern architectures often separate these pieces more flexibly. A typical CNN stacks several such modules:

$$\text{input} \to \text{conv block}_1 \to \text{conv block}_2 \to \cdots \to \text{classifier}.$$

At early layers, filters detect simple local features such as edges and color contrasts. At intermediate layers, receptive fields become larger and filters respond to textures, motifs, and object parts. At deeper layers, representations become more abstract and more task-specific.

This hierarchical organization is one of the central strengths of CNNs: local features are composed into increasingly complex ones.

11 Convolutions as sparse neural layers

A convolutional layer can be interpreted as a sparse alternative to a dense layer.

Suppose we have a one-dimensional input sequence $x_1, \ldots, x_T$ and a filter of width 3 with weights (a, b, c). The output at position t is

$$h_t = a\, x_{t-1} + b\, x_t + c\, x_{t+1}.$$

Compare this with a dense layer, where each output unit would depend on *all* input coordinates with distinct parameters. In convolution:

- each output depends only on a local neighborhood (sparse connectivity),
- the same weights (a, b, c) are reused at every position (parameter sharing).

These two constraints drastically reduce the number of parameters and encode the bias that the same local pattern should be detected everywhere.

12 Parameter counting in convolutional layers

Let the input be of size $H \times W \times C_{\text{in}}$, and let we use C_{out} kernels of size $K_h \times K_w \times C_{\text{in}}$. Then the number of trainable weights is

$$K_h K_w C_{\text{in}} C_{\text{out}},$$

plus usually one bias per output channel:

$$C_{\text{out}}.$$

Thus the total parameter count is

$$K_h K_w C_{\text{in}} C_{\text{out}} + C_{\text{out}}.$$

Notably, this number does *not* depend on the input height and width. This is a major contrast with dense layers, whose parameter count scales with the full input dimensionality.

Worked example — Parameter count in a convolutional layer

Suppose the input has $C_{\text{in}} = 3$ channels, and we use $C_{\text{out}} = 32$ filters of size 5×5. Then the number of kernel weights is

$$5 \cdot 5 \cdot 3 \cdot 32 = 2400.$$

Adding one bias per output channel yields

$$2400 + 32 = 2432$$

trainable parameters. Even if the input image were very large, the number of parameters would remain exactly the same.

This parameter efficiency is one of the main reasons CNNs are statistically and computationally attractive.

13 Receptive fields and depth

The *receptive field* of a unit is the region of the input that can influence that unit. In a CNN, receptive fields grow with depth. A unit in the first layer might depend on a 3×3 patch of the input, while a deeper unit may indirectly depend on a much larger region because it aggregates outputs of previous layers.

This is why stacking small kernels can be more effective than using one very large kernel. For example, two successive 3×3 convolutions yield an effective receptive field larger than a single 3×3 convolution, while introducing more nonlinearities and often fewer parameters than a single very large kernel.

14 Training CNNs

CNNs are typically trained by backpropagation, just like other neural networks. The key difference is that convolutional weights are shared across many spatial positions. Therefore, the gradient of one kernel coefficient is the sum of the contributions coming from every location where that coefficient was used.

If a kernel parameter w appears in multiple local dot products, then by the chain rule

$$\frac{\partial \mathcal{L}}{\partial w} = \sum_{\text{all locations}} \frac{\partial \mathcal{L}}{\partial a_{i,j}} \frac{\partial a_{i,j}}{\partial w}.$$

Thus gradient accumulation is a direct consequence of weight sharing.

Backpropagating gradients to the input of a convolution layer can be interpreted as a *transposed convolution* (sometimes called deconvolution in informal deep-learning terminology). This is not an inverse operation in general; it is the linear map corresponding to the transpose of the forward convolution operator.

15 Transposed convolution

If a forward convolution reduces spatial size, then the backward propagation of gradients must map smaller feature maps back to larger ones. This is implemented by the transpose of the linear operator induced by convolution. In modern deep learning, such an operator is also used directly as a learned upsampling layer and is then called a *transposed convolution*.

When stride is greater than one, transposed convolution conceptually inserts zeros between positions before applying a convolution-like operation. This explains why transposed convolution can increase spatial resolution.

Transposed convolutions are particularly important in decoder architectures for segmentation, image generation, and other dense prediction tasks.

16 Canonical CNN architectures

Modern CNN history is often taught through a sequence of landmark architectures. The details differ, but the progression reveals what each generation of models contributed.

16.1 LeNet

LeNet is the classical early CNN for grayscale image recognition. Its structure is simple and still pedagogically useful:

- alternating convolution and subsampling layers,
- relatively small filters (e.g. 5×5),
- fully connected layers at the end,
- originally sigmoid-like nonlinearities.

Its main significance is conceptual: it established the basic convolution–pooling hierarchy for image classification.

16.2 AlexNet

AlexNet marked the large-scale breakthrough of deep CNNs. Its main innovations include:

- successful training on large-scale image data,
- strong use of data augmentation,
- ReLU nonlinearities,
- dropout in dense layers,
- GPU-based training.

It showed that sufficiently deep convolutional models, trained properly, could dramatically outperform earlier vision systems.

16.3 VGG

VGG simplified CNN design by using repeated small 3×3 convolutions with stride 1 and occasional 2×2 max pooling. Its significance lies in standardization: instead of using many

heterogeneous filter sizes, it showed that deep stacks of small convolutions work extremely well. The cost is a large number of parameters, especially in the dense layers.

16.4 GoogLeNet / Inception

Inception architectures introduced the idea of computing filters of multiple spatial sizes in parallel and concatenating their outputs. A crucial trick was the use of 1×1 convolutions to reduce channel dimensionality before applying more expensive 3×3 or 5×5 kernels. This reduced both parameters and computation.

16.5 ResNet

ResNets introduced residual connections:

$$\mathbf{y} = F(\mathbf{x}) + \mathbf{x},$$

where $F(\mathbf{x})$ is the output of a convolutional block. This allows gradients to flow more directly through the network and made very deep CNNs trainable in practice.

16.6 DenseNet

DenseNets connect each layer to all subsequent layers within a dense block. This promotes feature reuse and further improves gradient flow.

17 Why 1×1 convolutions are useful

A 1×1 convolution does not aggregate spatial neighbors, but it mixes channels at each spatial position. If the input has C_{in} channels and the output has C_{out} channels, then a 1×1 convolution performs a learned linear projection

$$\mathbb{R}^{C_{\text{in}}} \to \mathbb{R}^{C_{\text{out}}}$$

independently at each location.

This is useful because it can:

- reduce the number of channels before more expensive convolutions,
- increase representational flexibility,
- add nonlinearity when followed by an activation function,
- implement channel-wise feature recombination.

18 Batch normalization

As networks become deeper, the distribution of activations seen by a layer can vary during training because upstream parameters change. Batch normalization addresses this by normalizing activations within a mini-batch, then applying a learned affine transformation.

For one activation dimension, given a mini-batch $\{h_i\}_{i=1}^{N_B}$, batch normalization computes:

$$\mu_B = \frac{1}{N_B} \sum_{i=1}^{N_B} h_i, \qquad \sigma_B^2 = \frac{1}{N_B} \sum_{i=1}^{N_B} (h_i - \mu_B)^2.$$

Then it normalizes:

$$\hat{h}_i = \frac{h_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}},$$

and finally rescales and shifts with learnable parameters γ, β:

$$h_i' = \gamma \hat{h}_i + \beta.$$

Worked example — Why batch normalization can preserve expressivity

At first sight, normalization might seem to constrain the representation too much. However, the learned scale and shift parameters restore flexibility:

$$h_i' = \gamma \hat{h}_i + \beta.$$

If the model needs a representation with mean m and standard deviation s, it can set

$$\beta = m, \qquad \gamma = s.$$

Thus batch normalization does not force the representation to remain standardized forever; rather, it stabilizes optimization while still allowing the network to learn the appropriate activation scale and offset.

Batch normalization often improves optimization speed and stability, especially in deep CNNs.

19 Residual learning

Residual connections are based on the idea that it may be easier for a network to learn a *residual correction* than a full transformation. Instead of learning directly

$$\mathbf{y} = G(\mathbf{x}),$$

a residual block learns

$$\mathbf{y} = F(\mathbf{x}) + \mathbf{x}.$$

This formulation is powerful because the identity path gives gradients a direct route through the network. Intuitively, if the best transformation is close to identity, the residual block only needs to learn the difference.

Residual learning is one of the key ideas that enabled the era of ultra-deep CNNs.

20 Advanced convolutional ideas

20.1 Causal convolutions

In sequential settings, one may want a convolution that does not access future inputs. A *causal convolution* restricts the receptive field so that output at time t depends only on positions up to t.

20.2 Dilated convolutions

A dilated convolution inserts gaps between kernel elements. If dilation factor is d, then a 2D dilated convolution can be written as

$$(\mathbf{x} \star_d \mathbf{K})(i,j) = \sum_m \sum_n \mathbf{x}(i + dm, j + dn)\, \mathbf{K}(m,n).$$

Dilated convolutions expand the receptive field without reducing feature-map resolution and without increasing the number of parameters.

This makes them attractive in dense prediction tasks such as semantic segmentation, where one wants large context but also accurate output localization.

21 CNNs for dense prediction and vision tasks

CNNs are not limited to image classification. The same building blocks appear in:

- **image classification:** predict a class label for the whole image,
- **image regression:** predict continuous attributes,
- **object detection:** localize and classify objects,
- **semantic segmentation:** assign a label to each pixel.

Dense prediction tasks require more than simply stacking convolutions and ending with fully connected classification layers. They often need:

- multi-scale context,
- upsampling or transposed convolutions,
- skip connections to preserve spatial detail,
- architectures specialized for localization accuracy.

Examples include deconvolutional segmentation networks, U-Nets with encoder–decoder structure and skip connections, and dilated-convolution architectures that preserve resolution while expanding context.

22 Putting a CNN together

A simple CNN for image classification often has the following overall shape:

$$\text{input} \to [\text{conv} + \text{ReLU} + \text{pool}]^{\times L} \to \text{flatten or global pooling} \to \text{dense classifier} \to \text{softmax}.$$

In modern architectures, flattening is often replaced by global average pooling, which reduces parameters and improves regularization.

A practical design workflow is:

1. decide the number of blocks and channel widths,
2. choose kernel sizes, strides, and padding,
3. track spatial dimensions across the network,
4. choose normalization and regularization methods,
5. decide whether dense layers, global pooling, or residual connections are appropriate.

Worked example — Tracking shapes through a simple CNN

Suppose the input is an RGB image of size

$$32 \times 32 \times 3.$$

Apply a convolutional layer with 32 filters of size 5×5, stride 1, and padding 0:

$$H' = W' = (32 - 5) + 1 = 28.$$

So the output is

$$28 \times 28 \times 32.$$

Apply 2×2 max pooling with stride 2:

$$H'' = W'' = \frac{28 - 2}{2} + 1 = 14.$$

So the pooled output is

$$14 \times 14 \times 32.$$

If the next convolution uses 64 filters of size 5×5, stride 1, and no padding, then

$$14 \to 10,$$

so the output is

$$10 \times 10 \times 64.$$

After another 2×2 max pooling with stride 2, the result is

$$5 \times 5 \times 64.$$

If we flatten it before a dense layer, the dense input dimension is

$$5 \cdot 5 \cdot 64 = 1600.$$

This kind of dimension bookkeeping is essential in CNN design.