Artificial Intelligence for Aerospace Engineers

Luca Magri,[1,2]

[1]Imperial College London, London, SW7 2AZ, UK
[2]Fellow of the Alan Turing Institute, London, NW1 2DB, UK

(LaTeX template inspired from annualreviews.org)

Keywords

Artificial intelligence, machine learning, optimization, data-driven methods, neural networks, aerospace engineering

Abstract

This is the first version of the notes I wrote in Sept 2021 for the course Artificial Intelligence for Aerospace Engineers at Imperial College London, Aeronautics Department. The course is open to a variety of students: 3^{rd}- and 4^{th}-year MEng students, as well as MSc students. The references cited in the notes are left for the keen readers, but they will not be examined. If you identify typos or mistakes (I am sure you will), please send me your corrections by e-mail: l.magri@imperial.ac.uk.

Contents

1. Lecture 5: Convolutional neural networks (CNNs)

Convolutional neural networks (CNNs) are (loosely) inspired by the "way we see". Similarly to feedforward neural networks, convolutional neural networks are made up of neurons that have learnable weights and biases. Differently from feedforward neural networks, each neuron receives some inputs, performs a dot product (filters the input) and activates the result. All the tips/tricks on neural networks still apply, however, CNNs differ from feedforward neural networks. This is because CNNs make the explicit assumption that the inputs are grid-like inputs (like images) as opposed to one-dimensional arrays (like in feedforward neural networks). This lecture contains some videos, the links of which can be found in the captions of some figures.

1.1. Objectives

The objectives of this lecture are

1. Introduce convolutional neural networks;
2. Analyse the components of CNNs (convolutional, pooling and fully connected layers);
3. Familiarize with practical design rules to build a CNN and train it.

Neocognitoron: The study on the visual cortex inspired the neocognitoron in the 1980s (Fukushima & Miyake 1982), which, in turn, inspired convolutional neural networks (LeCun et al. 1998).

1.2. Intuitive motivation

In this section, we explain a couple of motivations behind CNNs.

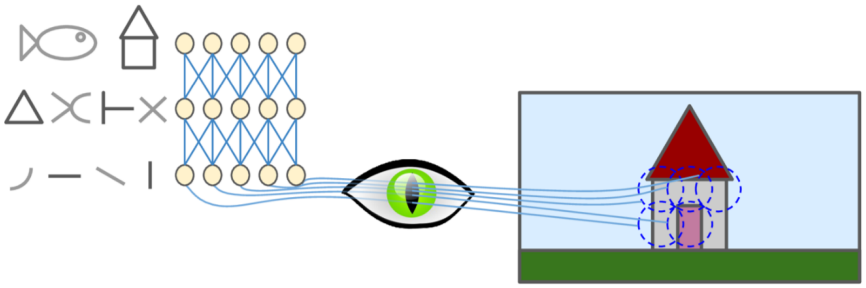

Figure 1: Biological neurons in the visual cortex respond to specific patterns in small regions of the visual field. These are called *receptive fields*. As the visual signal makes its way through consecutive brain modules, neurons respond to more complex patterns in larger receptive fields. Source: Géron (2019).

1.2.1. The neuroscience picture. Convolutional neural networks (CNNs) were inspired by the brain's visual cortex. CNNs have become popular in image recognition since the 1980s. In the 1950s, it was showed that many neurons in the visual cortex react only to visual stimuli located in a limited region of the visual field (Nobel-prize winning finding by David H. Hubel and Torsten Wiesel). In other words, neurons have a small local *receptive field*. The receptive fields of different neurons may overlap, and together they tile the whole visual field. For example, in Figure 1, local receptive fields of five neurons are represented by dashed circles.

As explained by Géron (2019), it was shown that some neurons react only to images of horizontal lines, whereas other neurons react only to lines with different orientations. Therefore, two neurons may have the same receptive field, but they can react ("get activated") to different line orientations. Finally, some neurons have larger receptive fields and react to more complex patterns, which are combinations of the simpler patterns. It was concluded that high-resolution neurons are based on the outputs of neighbouring lower-level neurons (in Figure 1 each neuron is connected only to a few neurons from the previous layer). This powerful architecture (the brain!) is able to detect incredibly intricate patterns in any area of the visual field.

1.2.2. The engineer's picture. Although CNNs are inspired by how the visual cortex works, it is best to think of CNNs as powerful machines that can approximate complicated functions with many variables that have spatial information, such as images, turbulent flow fields on a grid, etc.

1.3. Technical motivation

We explain the limitations of feedforward neural networks to introduce how CNNs are designed.

1.3.1. Recap on feedforward neural networks. Neural networks receive a flat input (i.e., one-dimensional array), which is transformed through a series of function compositions in hidden layers. Each hidden layer consists of a number of neurons. Each neuron is *fully connected* to all neurons in the previous layer. The last fully-connected layer is called the "output layer", which provides an estimate of the quantity of interest (a class in classification, or a continuous variable in regression).

1.3.2. Drawback of feedforward NNs for grid-like inputs. There are two main drawbacks of feedforward networks when they are applied to images or grid-like data. Let's consider an image in the grayscale with $28 \times 28 = 784$ pixels (Figure 2). We want to decrease the resolution to, say, 3×3. If we decide to design a feedforward network of 1 layer with 9 neurons, we would need $28 \times 28 \times 9 = 7056$ weights from the input layer to the hidden layer. That is a big number for a small image. Therefore, the first drawback is that *fully connected* layers, such as those of feedforward networks, are not suitable for images: they contain too many parameters.

Fully-connected networks do not scale well to large grid-like data, such as images.

Furthermore, if we want to use a feedforward network, we would need to *flatten* the 2D data into a one-dimensional array. In so doing, we would lose spatial information as to how the pixels are arranged. Therefore, the second problem of feedforward networks is that spatial correlations/dependencies are lost in the input data (Figure 3).

Spatial dependencies are lost in a feedforward neural network

Convolutional neural networks overcome both limitations.

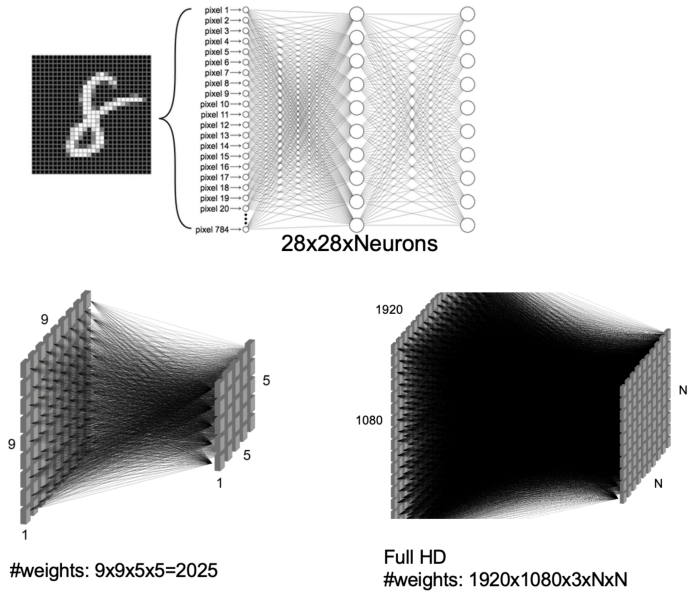

Figure 2: Fully connected layers in feedforward networks have a large number of parameters for images.

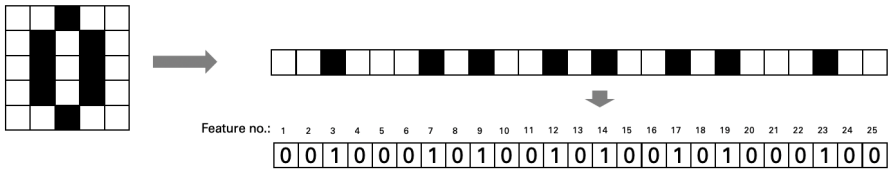

Figure 3: Spatial dependencies are lost after flattening the data in feedforward networks.

1.3.3. CNNs. CNNs exploit the data structure of the input, which consists of grid-like data points. Practically, CNNs constrain the architecture in a more principled and efficient way than feedforward NNs for grid-like data. In contrast to a feedforward NN, in which neurons in a layer are arranged in a one-dimensional array, the layers of a CNN have neurons arranged in three dimensions. Hence, every layer has a *width*, *height*, and *depth*.

Depth – Abuse of terminology: In CNNs, the word *depth* refers to the third dimension of a layer. In feedforward NNs, *depth* refers to the total number of layers in a network. Yes, this might be confusing.

1.4. Structure of a CNN

There are three types of layers, which are the building blocks of CNNs (Figure 4 for a sketch, Figure 13 for more details):

- **convolutional layer**, which includes the activation functions;
- **pooling layer**; and
- **fully-connected layer**.

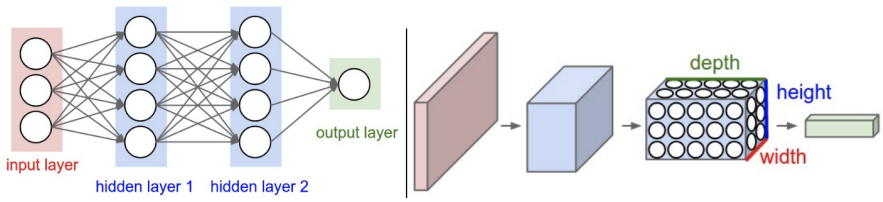

Figure 4: Left: Feedforward 2-hidden-layer neural network. Right: A CNN arranges its neurons in three dimensions (width, height, depth), as visualized in any layer. Every layer of a CNN transforms the 3D input volume into a 3D output volume of neuron activations. In this example, the red input layer is the image, therefore, the width and height are the dimensions of the image. The depth is 3 (Red, Green, Blue channels). Source: class CS231n (2021).

Example: Width, height, depth

As an example, let's consider CIFAR-10, which is a labelled subset of an 80 million images dataset, which is used to compare the model performance of new machine learning tools: https://www.cs.toronto.edu/~kriz/cifar.html. The input images of the CIFAR-10 dataset are an input volume with dimensions $32 \times 32 \times 3$ for width, height, depth, respectively. (Width and height are the pixels of one colour channel, whereas the depth corresponds to the three RGB colour channels.) The neurons in a layer are only connected to a small region of the previous layer, instead of all of the neurons in a fully-connected manner. This has dimensions $1 \times 1 \times 10$, because the CNN architecture reduces the full image into a single vector of class scores, arranged along the depth dimension (Figure 4).

1.4.1. Kernels and filters. In CNNs, a filter is a weighted average of the pixels of a receptive field. Practically, this is achieved by dot-multipliying a matrix (the filter) with the image. This operation starts from the top-left corner of an image. It is repeated by shifting the filter step by step to cover the entire image. This operation is known as *convolution* (Figs. 5,6,7). A 2D filter is also known as the *kernel*, which has nothing to do with the kernel trick in support vector machines. Therefore, a kernel is a 2D filter. If you use a 2D filter, you can call it "filter" or "kernel" interchangeably. The distinction between kernel and filter is important when you use 3D filters. In a 3D filter, you refer to it as the "filter" and you refer to the slices of it as to the "kernels" (Figure 7, bottom-right panel).

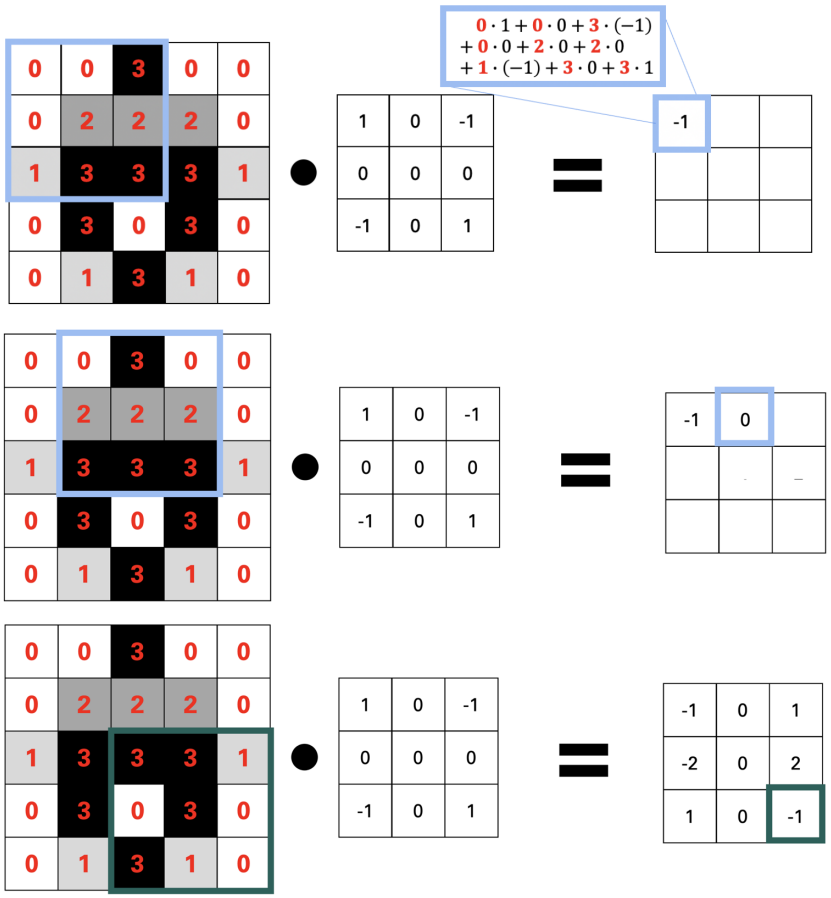

Figure 5: Example of a 2D filter (=kernel) that performs the *convolution operation*.

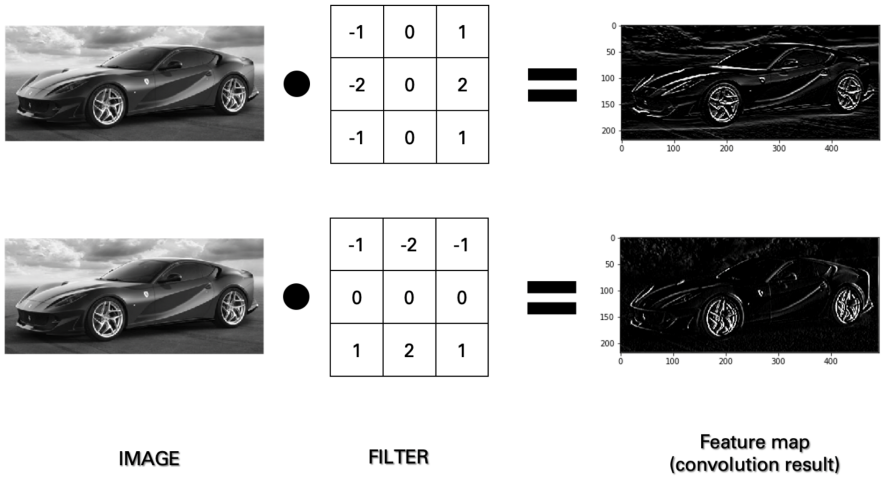

Figure 6: Example of 2D filters (=kernels) and convolution operations. Top: The filter selects horizontal features. Bottom: The filter selects vertical features.

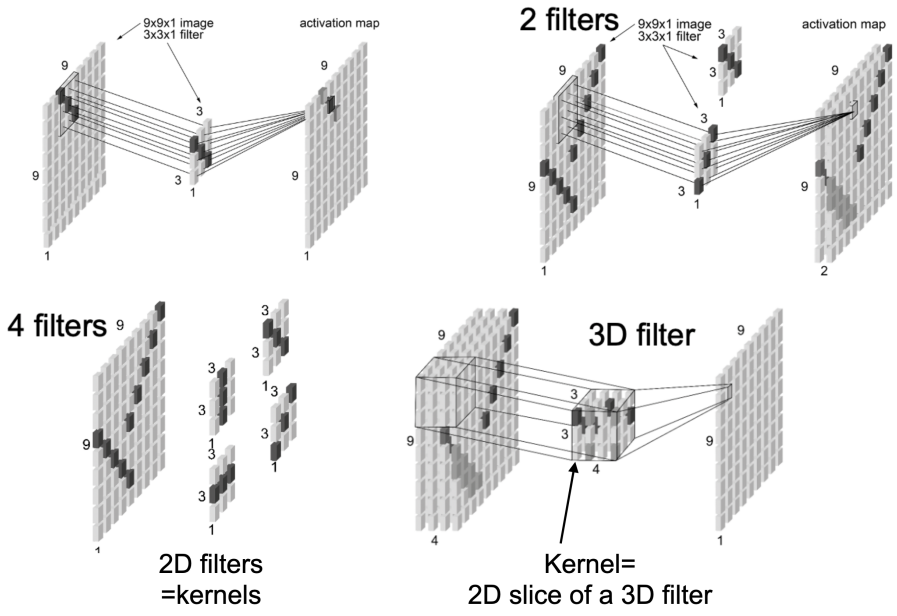

Figure 7: Different filters can be combined. If you use a 3D filter (bottom-right panel), each 2D slide of it is called *kernel*, which has nothing to do with kernels in support vector machines.

1.4.2. Convolutional layers. The convolutional layer is the core element of a CNN.

The convolutional layer's weights consist of a set of learnable filters.

Every filter is spatially small along width and height, but it fully extends through the depth of the input volume. During the forward pass, we slide (i.e., convolve) each filter across the width and height of the input volume and compute dot products between the entries of the filter and the input at any position. As we slide the filter over the width and height of the input volume, we produce a 2-dimensional feature map that gives a filtered input at every spatial position. Now, we have an entire set of filters in each convolutional layer, each of which produces a separate 2-dimensional feature map. We stack these feature maps along the depth dimension and produce the output volume.

Example: A typical filter on a first layer of a CNN might is $5 \times 5 \times 3$: 5 pixels for width and height, and 3 pixels for depth for the 3 colour channels.

Feature map: Also known as the activation map.

Convolutional layers have key properties:

- **Local connectivity.** Because we do not want to fully connect the layers, as explained in the introduction, we will connect each neuron to only a local region of the input volume. The spatial extent of this connectivity is a *hyperparameter* called

  the receptive field of the neuron, which is equal to the filter size.

  The connections are local along width and height, but they are full along the entire depth of the input volume.

- **Spatial arrangement.** Three *hyperparameters* control the size of the output volume (Figures 8, 9):
  - The **depth** of the output volume is the number of filters we decide to use. Each filter learns to look for a different feature in the input.
  - The **stride** is the step that the filter takes whilst sliding over the input. If we move the filters one pixel at a time, the stride is 1. If we move the filters two pixels at a time, the stride is two (Figure 8).
  - **Padding** adds numbers around the border of the input to control the spatial size of the output volume. This is because during the filtering, because of the stride and filter size, we lose information from the boundary. Typically, we use padding to exactly preserve the spatial size of the input volume, so that the input and output width and height are the same (Figure 8). Adding zeros as numbers is a common padding.

- **Parameter sharing.** So far we have introduced the convolution, but is the number of weight still acceptably small? If we do nothing, the number of weights is still too large. However, we do impose something: we share the parameters across the same feature map to regularize the problem.

  Neurons in each feature maps share the same weights and bias term.

1.4.3. Pooling layer. It is customary to periodically insert a pooling layer in-between successive convolutional layers in a CNN architecture. This is to reduce the spatial size of the

In a nutshell: The only difference between fully connected and convolutional layers is that the neurons in the convolutional layer are connected only to a local region in the input, and that many of the neurons in a convolution volume share the same parameters.

network, which reduces the number of parameters and computation to prevent overfitting. The pooling layer operates independently on every depth slice of the input and resizes it spatially. The typical pooling operation is the *max* operation. The most common form is a pooling layer with filters of size 2×2 applied with a stride of 2, which downsamples every depth slice in the input by 2 along both width and height, discarding 75% of the activations (assuming padding 1). Every max operation takes a max over 4 numbers (little 2×2 region in some depth slice). The depth dimension remains unchanged. A schematic of a CNN is shown in Figure 12.

1.4.4. Fully-connected layer. Neurons in a fully connected layer have full connections to all activations in the previous layer, exactly as in regular neural networks. Their activations can be computed with a matrix multiplication followed by a bias offset. See the Neural Network lecture.

1.4.5. Convolutional neural network. The full architecture is a combination of the aforementioned layers (Figures 10, 11, 12, 13).

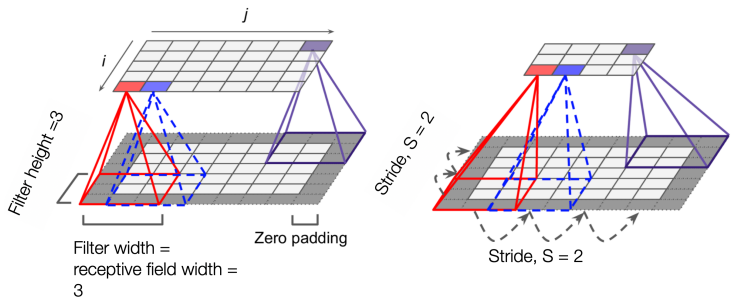

Figure 8: Left: Filter dimensions (receptive field) and padding. Right: stride. Adapted from (Géron 2019).

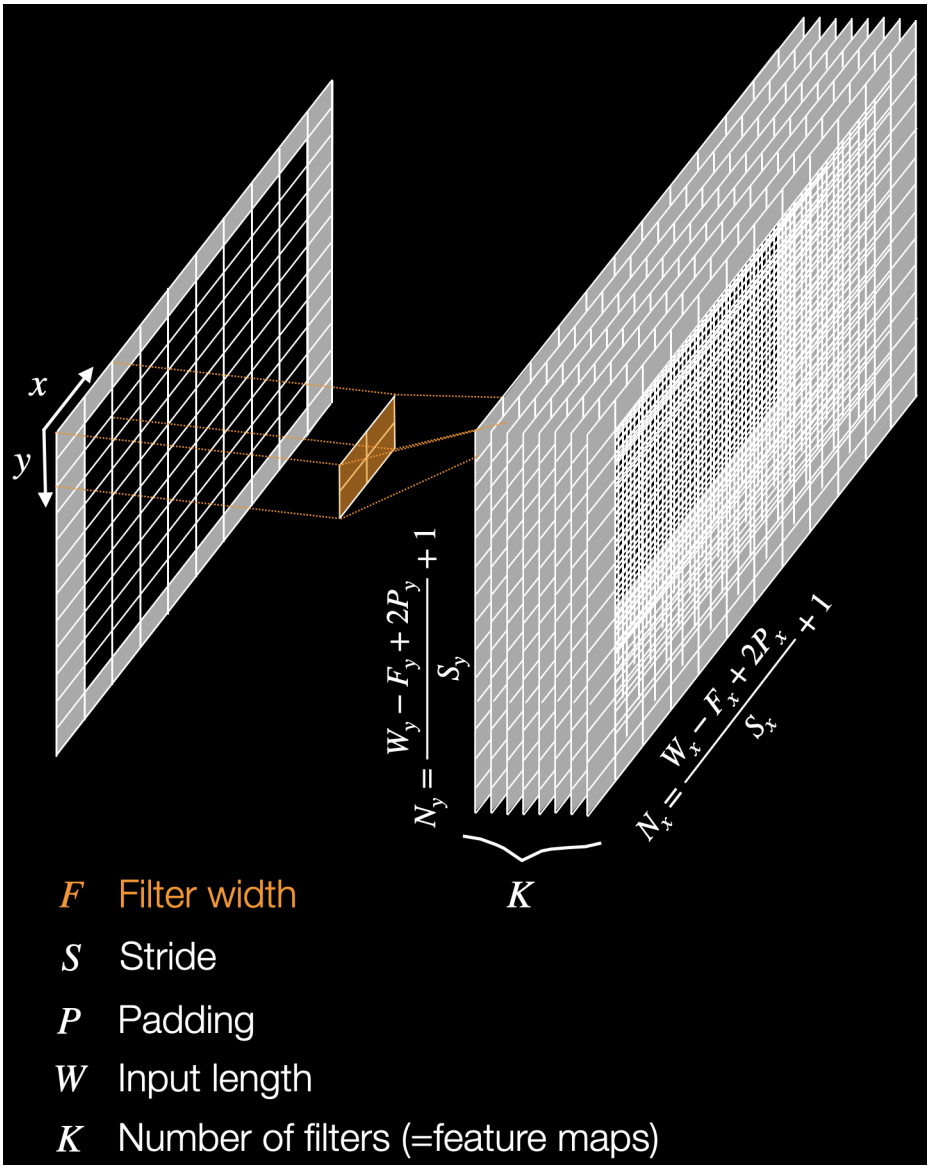

Figure 9: Key dimensions during convolution. See Section "How many neurons in a convolutional layer?". Video: https://vimeo.com/676842701/bd61034428.

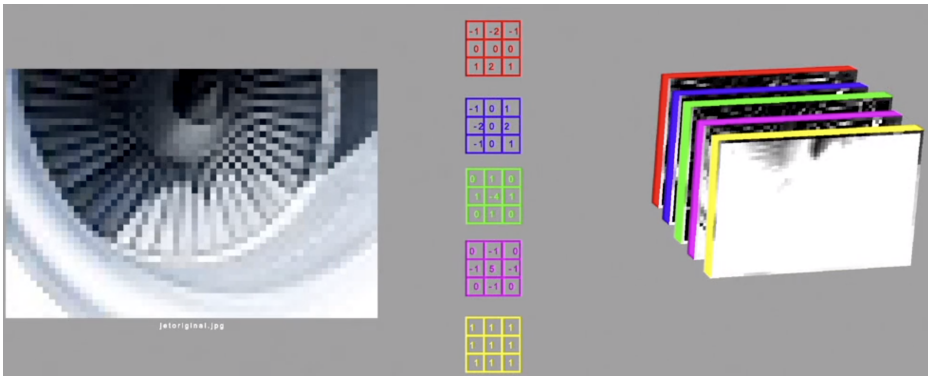

Figure 10: Example of convolution layer. Video: https://vimeo.com/678665862/917e325810.

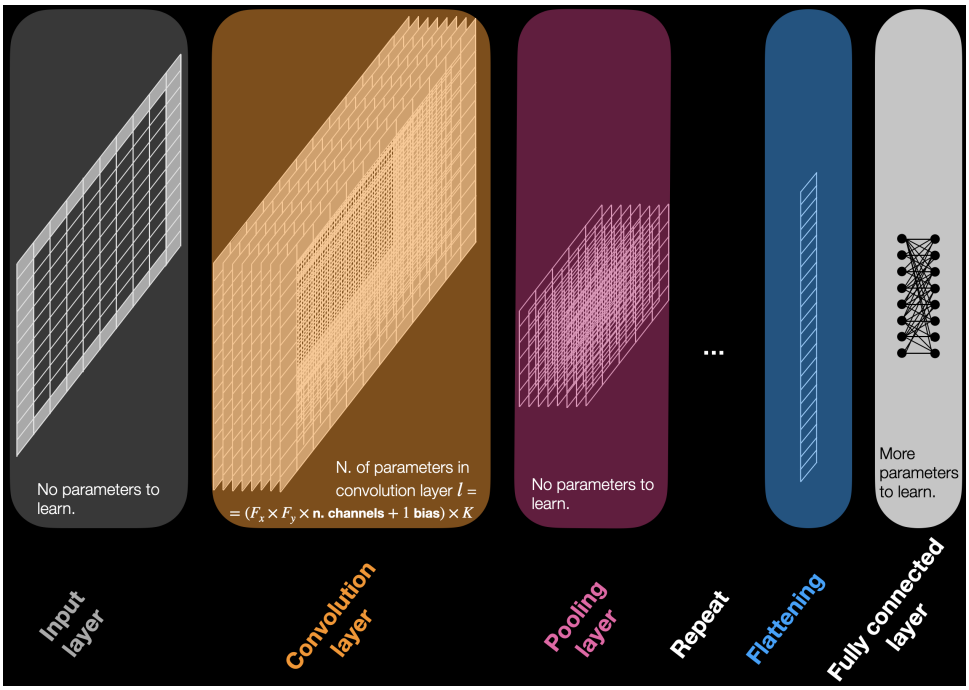

Figure 11: Schematic of a convolutional neural network with its components and parameters. Video: https://vimeo.com/676842701/bd61034428.

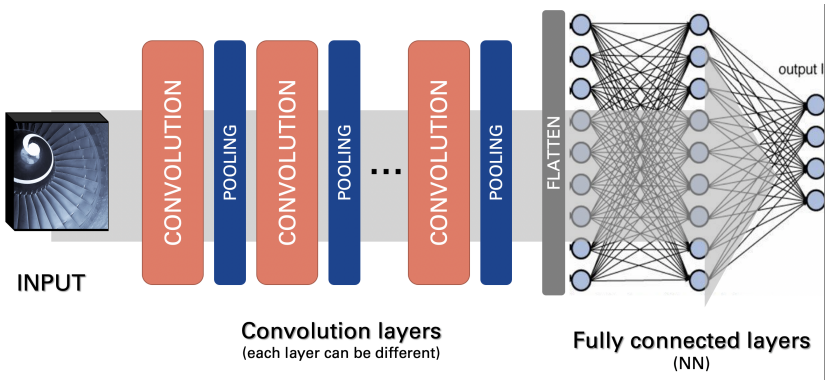

Figure 12: Example of a convolutional neural network with three convolution layers and a two-hidden-layer fully connected layer.

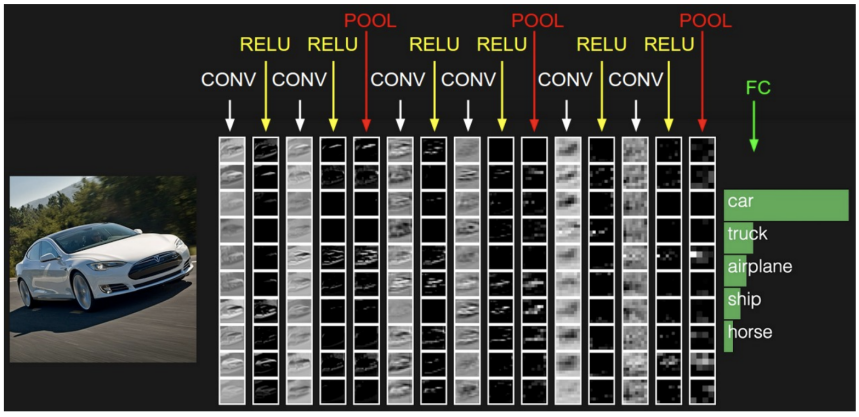

Figure 13: Example of a CNN architecture. Because it is difficult to visualize 3D volumes, each volume's slice is arranged in rows. Source: (class CS231n 2021).

Example of parameters of a CNN

Consider a simple CNN that takes $32 \times 32 \times 3$ images as inputs for scoring into 10 categories to classify (Figure 4).

Input $[32 \times 32 \times 3]$**.** This holds the raw pixel values of the image. In this example, the width is 32, the height is 32, and the depth is 3 (three colour channels R,G,B).

Convolutional layer $[32 \times 32 \times 12]$**.** This will compute the output of neurons that are connected to local regions in the input. This is done with a *filter*, which computes a dot product in a small region of the input to extract a feature (more details in the next sections). If we use 12 filters (it is up to us), this results in a volume of $32 \times 32 \times 12$.

Activation layer $[32 \times 32 \times 12]$**.** Typically, we choose a ReLU layer, which applies an elementwise ReLU activation function. This leaves the size of the volume unchanged ($[32 \times 32 \times 12]$), but crucially introduces the nonlinearity in the CNN. The activation layer is often included in the convolutional layer, but here we want to conceptually separate it for clarity.

Pooling layer $[16 \times 16 \times 12]$**.** This layer downsamples the convolutional layer along the spatial dimensions (width, height) through a pooling operation, such as max pooling. This reduces the convolutional layer to a smaller data set, for example, $[16 \times 16 \times 12]$. Of course, at this stage we lose some information because of the downsampling.

Fully-connected later $[1 \times 1 \times 10]$ This layer is a feedforward neural network, which computes the class scores. Each of the 10 numbers corresponds to a class score. Each neuron in this layer will be connected to all the numbers in the previous volume.

Therefore, a CNN is a stack of layers that transforms the image volume into an output volume. Each layer accepts an input 3D volume and transforms it to an output 3D volume. Each layer may or may not have weights (e.g. convolution and fully connected layers do, activation and pooling layers do not) and additional hyperparameters (e.g. convolutional, pooling and fully connected layers do).

How many neurons in a convolutional layer?

We can compute the spatial size of the output volume (i.e., the number of neurons) as a function of the input volume size ($W_x \times W_y$), the kernel size ($F_x \times F_y$), the stride (S_x and S_y), and the amount of padding (P_x and P_y) (Figure 9)

$$\text{Number of neurons in direction } d = \frac{W_d - F_d + 2P_d}{S_d} + 1, \qquad d = x, y. \tag{1.}$$

For example, for a 7×7 input ($W_x = W_y = 7$) and filter 3×3 ($F_x = F_y = 3$) with stride $S_x = S_y = 1$ and padding $P_x = P_y = 0$, we would have a 5×5 output (5 neurons in each direction plus 1 bias). With stride 2 we would have a 3×3 output (9 neurons in each direction plus 1 bias). A comprehensive analysis can be found in Section 8.2 in Aggarwal et al. (2018). A detailed analysis of the arithmetic of CNNs can be found in Dumoulin & Visin (2016).

How many parameters in a CNN?

In general, in a convolutional layer we have the following number of parameters (Figure 11)

$$\text{Number of parameters in layer } l = (F_x \times F_y \times \text{n. of channels} + 1) \times \text{n. of filters} \tag{2.}$$

The whole VGGNet, which is a CNN, is composed of convolutional layers that perform 3×3 convolutions with stride 1 and padding 1, and of pooling layers that perform 2×2 max pooling with stride 2 (and no padding). We can write out the size of the representation at each step of the processing and keep track of both the representation size and the total number of weights (Figure 14).

```
INPUT: [224x224x3]        memory:  224*224*3=150K   weights: 0
CONV3-64: [224x224x64]  memory:  224*224*64=3.2M   weights: (3*3*3)*64 = 1,728
CONV3-64: [224x224x64]  memory:  224*224*64=3.2M   weights: (3*3*64)*64 = 36,864
POOL2: [112x112x64]  memory:  112*112*64=800K   weights: 0
CONV3-128: [112x112x128]  memory:  112*112*128=1.6M   weights: (3*3*64)*128 = 73,728
CONV3-128: [112x112x128]  memory:  112*112*128=1.6M   weights: (3*3*128)*128 = 147,456
POOL2: [56x56x128]  memory:  56*56*128=400K   weights: 0
CONV3-256: [56x56x256]  memory:  56*56*256=800K   weights: (3*3*128)*256 = 294,912
CONV3-256: [56x56x256]  memory:  56*56*256=800K   weights: (3*3*256)*256 = 589,824
CONV3-256: [56x56x256]  memory:  56*56*256=800K   weights: (3*3*256)*256 = 589,824
POOL2: [28x28x256]  memory:  28*28*256=200K   weights: 0
CONV3-512: [28x28x512]  memory:  28*28*512=400K   weights: (3*3*256)*512 = 1,179,648
CONV3-512: [28x28x512]  memory:  28*28*512=400K   weights: (3*3*512)*512 = 2,359,296
CONV3-512: [28x28x512]  memory:  28*28*512=400K   weights: (3*3*512)*512 = 2,359,296
POOL2: [14x14x512]  memory:  14*14*512=100K   weights: 0
CONV3-512: [14x14x512]  memory:  14*14*512=100K   weights: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512]  memory:  14*14*512=100K   weights: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512]  memory:  14*14*512=100K   weights: (3*3*512)*512 = 2,359,296
POOL2: [7x7x512]  memory:  7*7*512=25K  weights: 0
FC: [1x1x4096]  memory:  4096  weights: 7*7*512*4096 = 102,760,448
FC: [1x1x4096]  memory:  4096  weights: 4096*4096 = 16,777,216
FC: [1x1x1000]  memory:  1000 weights: 4096*1000 = 4,096,000

TOTAL memory: 24M * 4 bytes ~= 93MB / image (only forward! ~*2 for bwd)
TOTAL params: 138M parameters
```

Figure 14: Example of number of parameters in a real CNN (VGGNet). Numbers are rounded off. Source: class CS231n (2021).

The output of a neuron in a convolutional layer

Let's recap. In each feature map, there is one neuron per pixel. All neurons within the same feature map share the same weights and biases. Neurons in different feature maps have different weights and biases. The output of a neuron is given by a daunting mathematical expression (Figure 15 for a visualization)

$$z_{ij,k} = a\left(\sum_{u=0}^{F_x-1}\sum_{v=0}^{F_y-1}\sum_{k'=0}^{K-1} b_k + x_{(iS_x+u)\ (jS_y+v),\ k'} \cdot \theta_{(iS_x+u)\ (jS_y+v),\ k',\ k}\right) \quad \text{for layer } l. \tag{3.}$$

where z_{ijk} is the output of the neuron located in row i, column j in the feature map k of the convolutional layer l; $S_{(\cdot)}$ is the stride; $F_{(\cdot)}$ is the size of the receptive field (kernel); K is the number of feature maps in the previous layer (l–1); $x_{i'j'k'}$ is the output of the neuron located in layer $l-1$, row i', column j', feature map k' (or channel k' if the previous layer is the input layer); b_k is the bias term for feature map k in layer l (you can think of it as a shift in the overall brightness of the feature map); and $\theta_{uvk'k}$ is the connection weight between any neuron in feature map k of the layer l and its input located at row u, column v (relative to the neuron's receptive field), and feature map k'.

Memory

Consider a convolutional layer with 5×5 filters, outputting 200 feature maps of size 150×100, with stride 1 and padding 1. If the input is a 150×100 RGB image (three channels), then the number of parameters is $(5 \times 5 \times 3 + 1) \times 200 = 15200$ ($+\ 1$ corresponds to the bias term), which is fairly small compared to a fully connected layer. However, each of the 200 feature maps contains 150×100 neurons, and each of these neurons needs to compute a weighted sum of its $5 \times 5 \times 3 = 75$ inputs. This gives 225 million float multiplications. If the feature maps are represented using 32-bit floats, then the convolutional layer's output will occupy $200 \times 150 \times 100 \times 32 = 96$ million bits (12 MB) of RAM. If a training batch contains 100 instances, then this layer uses up 1.2 GB of RAM.

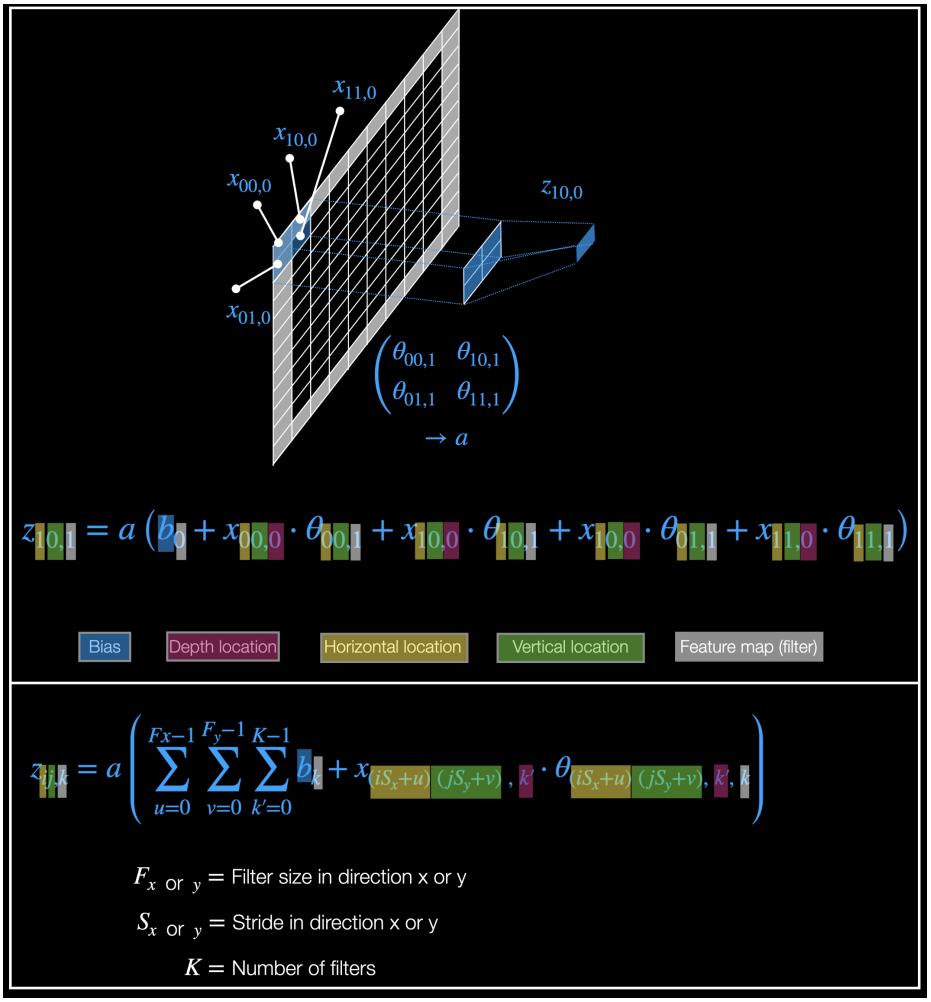

Figure 15: Visualization of the neuron's equation. Video: https://vimeo.com/678665280/1af7c1421f.

1.5. Practical design

- **Input layer.** The input layer that contains the image should be divisible by 2 many times. Common numbers include 32 (e.g. CIFAR-10), 64, 96 (e.g. STL-10), or 224 (e.g. common ImageNet CNNs), 384, and 512.

- **Filter, stride, padding.** Typical sizes are 3×3 or 5×5 with stride of 1 and padding such that the dimension of the convolutional layer is the same as the input. Note that $P = (F - 1)/2$ preserves the input size.

- **Pooling.** The most common setting is to use max-pooling with 2×2 receptive fields (i.e. $F = 2$), and with a stride of 2 (i.e. $S = 2$). This discards exactly 75% of the activations in an input volume (due to downsampling by 2 in both width and height, assuming padding 1).

- **Double the number of filters after each pooling.** This is a rule of thumb, but, if you have no clue, this could be a good starting point.

- **Augment the data**. This is a regularization technique to prevent overfitting. Artificially increase the data size by generating realistic variations of the training data points: shift, rotate, risize, contrast are common strategies. This forces the model to be less sensitive to variations in the position, orientation, size, and lighting.

- **Dropout.** This is another common regularization technique to prevent overfitting. At every training step, p% of neurons (except for the output layer) are ignored during this training step, but they may be active during the next step. The hyperparameter p is called the dropout rate, which is typically set to 40–50% in convolutional neural networks. After training, no neuron is dropped out, thus, all neurons participate in the prediction.

1.6. Exercises

1. How many parameters would the following CNN have with and without parameter sharing?

   Consider the AlexNet architecture (Figure 16), which accepts images of size $[227 \times 227 \times 3]$. On the first Convolutional Layer, it uses neurons with receptive field size $F = 11$, stride $S = 4$ and no padding $P = 0$. The convolutional layer has 96 feature maps.

   $(227 - 11)/4 + 1 = 55$ is the the number of neurons in each dimension,. Hence, the convolutional layer output volume has size $[55 \times 55 \times 96]$. Each of the $55 \times 55 \times 96$ neurons in this volume is connected to a region of size $[11 \times 11 \times 3]$ in the input volume. All 96 neurons in each depth column are connected to the same $[11 \times 11 \times 3]$ region of the input, but with different weights. *Without parameter sharing*, the number of neurons is $55 \times 55 \times 96 = 290400$ neurons, with each having $11 \times 11 \times 3 = 363$ weights plus 1 bias. Together, this adds up to $290400 \times 364 = 105705600$ parameters. It is a big number. *With parameter sharing*, we constrain the neurons in each depth slice to use the same weights and bias. Hence, the first convolutional layer would have only 96 unique set of weights (one for each depth slice), for a total of $96 \times 11 \times 11 \times 3 = 34848$ unique weights, or $34848 + 96 = 34944$ parameters including the 96 biases. In other words, all 55×55 neurons in each depth slice use the same parameters.

2. Read about excellent CNN architectures (AlexNet, GoogleNet, VGGNet, ResNet) in chapter 14 of Géron (2019).

Layer	Type	Maps	Size	Kernel size	Stride	Padding	Activation
Out	Fully connected	–	1,000	–	–	–	Softmax
F9	Fully connected	–	4,096	–	–	–	ReLU
F8	Fully connected	–	4,096	–	–	–	ReLU
C7	Convolution	256	13 × 13	3 × 3	1	`same`	ReLU
C6	Convolution	384	13 × 13	3 × 3	1	`same`	ReLU
C5	Convolution	384	13 × 13	3 × 3	1	`same`	ReLU
S4	Max pooling	256	13 × 13	3 × 3	2	`valid`	–
C3	Convolution	256	27 × 27	5 × 5	1	`same`	ReLU
S2	Max pooling	96	27 × 27	3 × 3	2	`valid`	–
C1	Convolution	96	55 × 55	11 × 11	4	`valid`	ReLU
In	Input	3 (RGB)	227 × 227	–	–	–	–

Figure 16: AlexNet architecture (Krizhevsky et al. 2012). Source: Géron (2019).

1.7. Assignments 3 and 4

These need to be submitted on Vocareum. For assignment 3 (deep neural networks) the deadline for submission is 2pm on Tuesday 1st of March. For assignment 4 (CNNs) the deadline for submission is 2pm on Tuesday 8th of March. No extensions will be allowed.

LITERATURE CITED

Aggarwal CC, et al. 2018. Neural networks and deep learning. *Springer* 10:978–3

class CS231n C. 2021. Stanford CS class CS231n: Convolutional Neural Networks for Visual Recognition. `https://cs231n.github.io/convolutional-networks/#layer`. [Online; accessed 21/1/22]

Dumoulin V, Visin F. 2016. A guide to convolution arithmetic for deep learning. *arXiv preprint arXiv:1603.07285*

Fukushima K, Miyake S. 1982. Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition. In *Competition and cooperation in neural nets*. Springer, 267–285

Géron A. 2019. Hands-on machine learning with scikit-learn, keras, and tensorflow: Concepts, tools, and techniques to build intelligent systems. O'Reilly Media

Krizhevsky A, Sutskever I, Hinton GE. 2012. Imagenet classification with deep convolutional neural networks, In *Advances in Neural Information Processing Systems*, eds. F Pereira, CJC Burges, L Bottou, KQ Weinberger, vol. 25. Curran Associates, Inc.

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