

Convolutional Neural Networks Demystified: A Matched Filtering Perspective Based Tutorial

Ljubiša Stanković, *Fellow, IEEE*, Danilo Mandić, *Fellow, IEEE*,

Abstract

Deep Neural Networks (DNN) and especially Convolutional Neural Networks (CNN) are a de-facto standard for the analysis of large volumes of signals and images. Yet, their development and underlying principles have been largely performed in an ad-hoc and black box fashion. To help demystify CNNs, we revisit their operation from first principles and a matched filtering perspective. We establish that the convolution operation within CNNs, their very backbone, represents a matched filter which examines the input signal/image for the presence of pre-defined features. This perspective is shown to be physically meaningful, and serves as a basis for a step-by-step tutorial on the operation of CNNs, including pooling, zero padding, various ways of dimensionality reduction. Starting from first principles, both the feed-forward pass and the learning stage (via back-propagation) are illuminated in detail, both through a worked-out numerical example and the corresponding visualizations. It is our hope that this tutorial will help shed new light and physical intuition into the understanding and further development of deep neural networks.

I. CONVOLUTIONAL NEURAL NETWORK – CNN

We live in a world overwhelmed by data with multiple sources routinely generating high resolution signal/image streams. Processing such data comes with an inevitable bottleneck of the (curse of) dimensionality. To put this into perspective, even a modest resolution 640×480 VGA image comprises 307,200 pixels, while a 1920×1080 HDTV image contains 2,073,600 pixels; if those images are processed using neural networks (NN), then each pixel requires one neuron at the NN input layer. This is followed by at least one hidden layer, so that even for a typical small-scale fully connected hidden layer with, for example, 1024 nodes, the number of parameters quickly becomes prohibitive [1], [2], [3].

In practical applications, this issue is partially mitigated by exploiting the fact that most physical data sources exhibit a smooth nature, so that the neighboring signal points or image pixels exhibit some sort of similarity. This allows us to employ local information in the form of features, which describe the analyzed signals/images; our task then becomes to search for specific localized features in data, instead of the standard brute force approaches which scale exponentially with the data volume.

Another advantage of operating in the feature space, instead of with the raw pixels, is that this resolves the important problem of position change of the patterns in data due, for example, to translation. Namely, if a certain data feature changes its position, then a pixel-wise approach will assume a complete change in pixels, while a feature-wise approach will look for specific shapes anywhere in the signal.

Similar reasoning also underpins the operation of convolutional neural networks (CNNs), which boils down to performing some sort of search for features in the analyzed signal, such that these features are invariant to their position change [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16]. More specifically, the window used in convolution within CNNs (*convolution filter or convolution kernel*) is capable of recognizing precisely one feature that is matched to its form. To do so, we perform feature matching over the whole signal, akin to a mathematical lens in search of specific forms. For more details on a similar approach to graph convolutional neural networks, please see our sister paper [17].

II. PRINCIPLE OF MATCHED FILTERING

While the use of a convolutional window has become a de facto standard in CNNs, an open question remains of how we can justify that the convolution is an appropriate operation for detecting features in a signal – a subject of this tutorial. To this end, we draw inspiration from the matched filter theory, whereby the convolution of the considered signal with the feature that we are looking will confirm the existence and location of the feature at hand. Recall that the output of a matched filter is calculated through a convolution [18]

$$y(n) = x(n) * w(-n) = \sum_m x(m)w(m-n) = \sum_m x(n+m)w(m) = x(n) * _c w(n),$$

where $w(n)$ denotes the feature that we are looking for in the analyzed signal $x(n)$ and $*_c$ denotes the convolution with the time-reversed feature/template, $w(-n)$, which serves as the “impulse response”. Therefore, the best search function to detect a feature, $w(n)$, within a signal $x(n)$, would be through a convolution of the signal $x(n)$ with $w(-n)$.

Remark 1: Convolution based feature detection produces a result that is independent of the feature position within the considered signal, since $y(n) = x(n) * w(-n)$ is calculated by sliding the window (filter/kernel), $w(n)$, and multiplying it with the

signal segments, $x(n)$ for all n . The same holds for an image, whereby when calculating the corresponding convolution with $w(-m, -n)$ a two-dimensional filter $w(m, n)$ slides along the image in both spatial directions.

If we are looking for one of K features in the input signal then we can form a bank of K matched filters with outputs

$$y_k(n) = x(n) * w_k(-n) = \sum_m x(n+m)w_k(m) = x(n) *_c w_k(n), \quad k = 1, 2, \dots, K.$$

The decision about the feature k which is contained in the input signal is based on

$$k = \arg\{\max\{x(n) *_c w_1(n), x(n) *_c w_2(n), \dots, x(n) *_c w_K(n)\}\}.$$

Example 1. The principle of the matched filter is illustrated on two noisy signals shown in the two top panels in Fig. 1. The features contained in these signals are given in the two middle panels in Fig. 1 and are designated by the red and blue lines. Both signals are convolved with the reversed versions of these two features (serving as the matched filter impulse responses) according to $y(n) = x(n) * w(-n)$. The outputs of the matched filters (red and blue filter) are shown in the bottom panels in Fig. 1. The left two panels at the bottom show the output of the red and blue matched filter to the first input signal at the top-left, while the right two panels at the bottom show the output of the red and blue matched filter to the second input signal at the top-right. We can conclude that the first input signal contains the red feature (since the output is above the threshold line), while the second input signal contains the blue feature.

Notice that in the definition of the matched filter, the convolution $x(n) * w(-n)$ corresponds to the cross-correlation of $x(n)$ and $w(n)$ rather than their convolution, $x(n) * w(n)$. This because we have used a digital filter to implement convolution, which has been made possible by the impulse response being a time-reversed version of the feature. Nonetheless, the network is called the *convolutional neural network* (CNN) rather than the *cross-correlational neural network*, with all notations assuming that the convolution is applied after one of the signals is time reversed, that is $x(n) * w(-n)$. This is implicitly indicated in various notations in literature, for example, $\mathbf{x} * \text{rot180}(\mathbf{w})$ or $\text{conv}(\mathbf{x}, \text{reverse}(\mathbf{w}))$. We will use a simplified notation $\mathbf{x} *_c \mathbf{w}$, to indicate that the second signal in the convolution is reversed.

Remark 2: Consider receiving a waveform which is one from a set of possible waveforms (dictionary) for our problem. The task is to determine which of the template waveforms it matches best. Then, it intuitively makes sense to compute the correlation of the received waveform against each member of the alphabet. Assuming the same normalized energy, the maximum correlation occurs against the correct template waveform from the dictionary. One way of calculating this correlation is by putting the received waveform through a bank of filters with each having as an impulse response one of the alphabet signals, time reversed. Then, the maximum output value will be equal to the cross-correlation of the received signal with the alphabet signal.

CNNs are a type of neural network that use convolution layers, which consist of a set of convolutional filters. Convolutional filters are typically applied over different layers, each aiming to identify a different feature in a signal. By learning different forms of the feature space, convolutional networks allow for robust, efficient analysis and classification of signals and images.

III. THE FORWARD PROPAGATION PATHWAY IN CNNs

We shall now use a matched filter perspective to shed a new light on key algorithmic steps in the operation of CNNs, for simplicity we assumed already (initialized) or calculated weights of the convolutional filters (*forward propagation*). The weight update will be addressed afterwards.

- 1) **Input:** Consider a signal, \mathbf{x} , with N samples, given by

$$\mathbf{x} = [x(0), x(1), \dots, x(N-1)]^T$$

(or an image of $N \times N$ samples) as the input to a neural network.

A common goal in CNNs is to classify input signals (images) into several non-overlapping sets (clusters).

- 2) **Convolution layer:** This operation employs a convolutional filter of M elements (*cf.* a filter of $M \times M$ samples for images). The convolution layer filter is sometimes called convolutional kernel. Common choices are, for example, those of length $M = 3$ or $M = 5$. Note that K such filters are applied, if we are looking for K features in \mathbf{x} . The elements of the k -th response of the first convolutional layer are then

$$\mathbf{w}_k^1 = [w_k^1(0), w_k^1(1), \dots, w_k^1(M-1)]^T,$$

for $k = 1, 2, \dots, K$. Then, the output signals are

$$\mathbf{y}_k^1 = \mathbf{x} *_c \mathbf{w}_k^1,$$

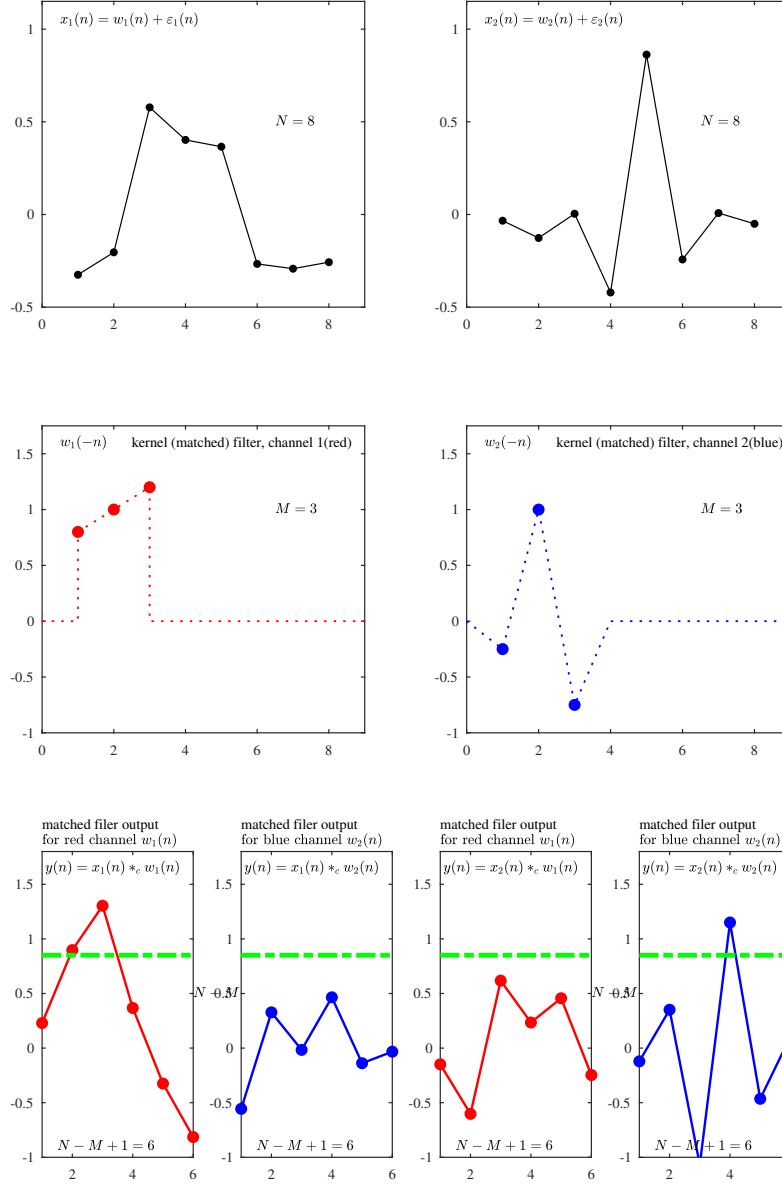


Fig. 1. Illustration of the matched filter operation. Top panels: Two noisy input signals of the length $N=8$ (noise is denoted by $\varepsilon(n)$). Middle panels: Two features that we are searching for in the input signals of the length $M=3$. The reversed versions of these features serve as the inputs to the corresponding (red and blue channels) matched filters (middle panels). Bottom panels: Deep Neural Networks (DNN) and especially Convolutional Neural Networks (CNN) are a de-facto standard for the analysis of large volumes of signals and images. Yet, their development and underlying principles have been largely performed in an ad-hoc and black box fashion. To help demystify CNNs, we revisit their operation from first principles and a matched filtering perspective. We establish that the convolution operation within CNNs, their very backbone, represents a matched filter which examines the input signal/image for the presence of pre-defined features. This perspective is shown to be physically meaningful, and serves as a basis for a step-by-step tutorial on the operation of CNNs, including pooling, zero padding, various ways of dimensionality reduction. Starting from first principles, both the feedforward pass and the learning stage (via backpropagation) are illuminated in detail, both through a worked-out numerical example and the corresponding visualisations. It is our hope that this tutorial will help shed new light and physical intuition into the understanding and further development of deep neural networks. The outputs of the read and blue matched filter to the each of input signals, with appropriate decision line, used to decide which feature is contained in the corresponding input signal.

where \ast_c denotes the convolution of the time-reversed filter (channel), \mathbf{w}_k^1 , and the signal, \mathbf{x} , (cross-correlation). For further illustration, the element-wise form of this convolution, for $M = 3$, is given by

$$y_k^1(n) = w_k^1(0)x(n) + w_k^1(1)x(n+1) + w_k^1(2)x(n+2) = \sum_{m=0}^{M-1} w_k^1(m)x(n+m). \quad (1)$$

The dimension of the k th output, \mathbf{y}_k^1 , is $(N - M + 1) \times 1$. The last element in $y_k^1(n)$ is obtained for $(n + m) = N - 1$ with $m = M - 1$, that is, $y_k^1(N - M)$. For $M = 3$, the last element in $y_k^1(n)$ is $y_k^1(N - 3)$.

In total, K such output signals of the convolution layer, \mathbf{y}_k^1 , $k = 1, 2, \dots, K$, are obtained, with the total number of the output signal elements, $y_k^1(n)$, from the first convolution layer therefore being $K(N - M + 1)$.

Remark 3: The total number of filter weights, $w_k^1(n)$, in the first convolutional layer is equal to the product of the convolution filter length, M , and the number of filters, K , that is MK . This is typically much smaller than in the case of a fully connected neural network, whereby each of N input signal samples is connected through weights to each of K output signals, to yield NK connections.

If an image is considered, then the output image of the convolutional filter is of the size $(N - M + 1) \times (N - M + 1)$. There are K such images and the total number of the filter weights is KM^2 , which is again smaller than KN^2 connections in the standard fully connected layer.

Example 2. Relation to standard neural networks. The input-output relation for the CNN simplifies into standard neural network as a special case. This is immediately seen by first considering the element-wise form of the output at an unindexed neuron (before the activation function), given by

$$y(n) = \sum_{k=1}^N w_k x_k(n) \quad (2)$$

where $x_k(n)$ designates the input to the neuron k in a time instant, n .

In the CNN, one signal, $x(n)$, is considered as the input whose N samples arrive simultaneously at N input neurons (these are considered as one training datum). Such data are connected to K output neurons in the first convolutional layer, so that the input-output relation within the CNN framework becomes

$$y_k = \sum_{m=0}^{N-1} w_k^1(m)x(m) = w_k^1(0)x(0) + w_k^1(1)x(1) + \dots + w_k^1(N-1)x(N-1). \quad (3)$$

Next, by comparing (2) and its CNN notation in (3) to (1) we can conclude that the standard neural network is a **special case** of the CNN, with $M = N$. Namely, the relation in (1), for $M = N$ is of the form

$$y_k^1(n) = w_k^1(0)x(n) + w_k^1(1)x(n+1) + w_k^1(2)x(n+2) + \dots + w_k^1(N-1)x(n+N-1).$$

If zero-padding is not performed for the input signal, this relation can be calculated only for $n = 0$ (since $x(n)$ is defined only for $n = 0, 1, \dots, N-1$), to yield

$$y_k^1 = y_k^1(0) = w_k^1(0)x(0) + w_k^1(1)x(1) + w_k^1(2)x(2) + \dots + w_k^1(N-1)x(N-1), \text{ for } k = 1, 2, \dots, K, \quad (4)$$

so that we arrived at (3). For this reason, $M \ll N$ is typically used in CNNs. See Fig. 2 for a step-by-step illustration of the operation of the first convolutional layer.

Note: All results derived next for the convolutional layer also hold for the standard, fully connected layer, with $y_k^1 = y_k^1(n)$, and using only $n = 0$, and $M = N$.

In the standard neural network, when the input $x(n)$, $n = 0, 1, \dots, N-1$, is simultaneously applied to N input neurons that are connected to K output neurons, with y_k , $k = 1, 2, \dots, K$, the number of different weights, $w_k^1(n)$, is NK , which is larger than MK , the number of weights in the CNN.

3) **Bias:** In the convolution layer (like in standard neural network layers), the bias (constant) term may be added, to yield

$$y_k^1(n) = \sum_{m=0}^{M-1} w_k^1(m)x(n+m) + b_k^1.$$

The vector form of the CNN output then becomes

$$\mathbf{y}_k^1 = \mathbf{x} \ast_c \mathbf{w}_k^1 + b_k^1,$$

with the total number of coefficients into every convolution increased by one.

Remark 4: The number of weights (parameters) in a CNN depends only on the size of the convolutional (feature matching) filter. For time-domain signals, with the bias term included, the total number of weights is $K(M + 1)$. For an image, the total number of weights is $K(M^2 + 1)$.

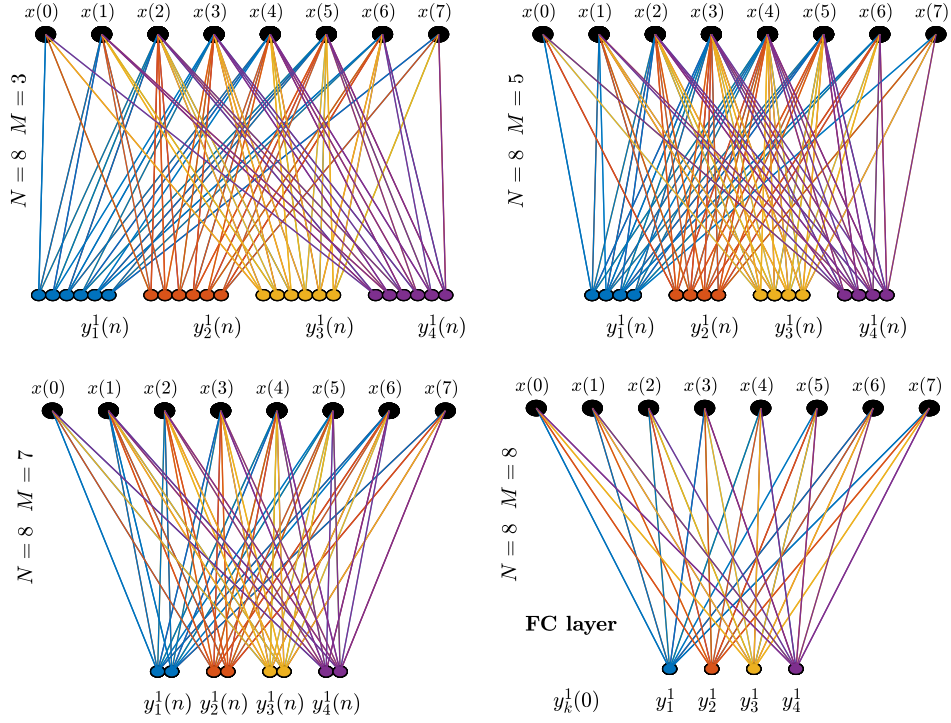


Fig. 2. Operation of a Convolutional Neural Network for an input of size N , convolution layer with $K = 4$ output neurons, and for various lengths, M , of the convolution filter. For $M = N$, the convolutional layer reduces to the fully connected (FC) layer. Top left: The case for $N = 8$ and $M = 3$. Top right: The case for $N = 8$ and $M = 5$. Bottom left: The case for $N = 8$ and $M = 7$. Bottom right: The case for $N = 8$ and $M = 8$, corresponding to the FC layer.

- 4) **Zero-Padding:** In some cases, the output of the convolution stage should be of the *same* size as the input signal (image), instead of the minimum of $(N - M + 1)$ which results from the convolution. This can be achieved if the input signal (image) is zero-padded with an appropriate number of zeros. For example, for $M = 3$ and any N , we could add a zero at $x(-1) = 0$ and a zero at $x(N) = 0$, to calculate the convolution output $y(n)$ as

$$y_k^1(n) = w_k^1(0)x(n-1) + w_k^1(1)x(n) + w_k^1(2)x(n+1).$$

which yields the same number of samples as in the input signal, $x(n)$. Namely, now we can calculate $y(0)$ as the first element in the convolution, and $y(N-1)$ as the last element in the convolution.

Remark 5: In general, if the filter response length is M , the signal should be padded with $(M-1)$ zeros to perform a convolution of length N , as the length of the convolution operation is $N - M + 1$, if the filter response should not be moved outside the signal. For a filter with an odd number of elements M , if we desire that the convolution result has the same number of samples as the input signal, the input signal could symmetrically be zero-padded, by $(M-1)/2$ elements before the starting sample at $n = 0$, and with the same number of zero elements after the signal sample at $n = N-1$.

- 5) **Nonlinear activation function:** Signals and images are far from exhibiting a linear nature, while convolution (correlation) is a linear operation. To this end, a non-linearity is applied to the output of a convolutional layer. Such a nonlinear map will restrict the output values to reside within a specified output range, as in the case of sigmoid type of nonlinearities. The most common nonlinear activation function for CNNs is the Rectified Linear Unit (ReLU), defined by

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

In the CNNs, this function has several advantages over sigmoidal-type activation functions: (i) It does not saturate for the positive values of its input thus producing nonzero gradient for large input values, (ii) Its calculation is not computationally demanding, and (iii) In practical applications ReLU converges faster than the saturation-type nonlinearities (logistic, tanh). Moreover, this function does not activate all neurons at the same time, so that sparsification by deactivation is achieved for each neuron producing negative value as an input to the ReLU activation function.

The output of one convolutional layer, after the activation function, then becomes

$$f(\mathbf{y}_k^1) = f(\mathbf{x} * \mathbf{w}_k^1 + b_k^1) = f(\mathbf{w}_k^1, \mathbf{x}).$$

For our example with $M = 3$, the element-wise output is therefore

$$f(y_k^1(n)) = f(w_k^1(0)x(n) + w_k^1(1)x(n+1) + w_k^1(2)x(n+2) + b_k^1).$$

Example 3. Consider a multivariate signal $\mathbf{y}_k = \mathbf{x} * \mathbf{w}_k^1 + b_k$, with $K = 3$ convolution filters (channels), $k = 1, 2, 3$ given by

$$\begin{aligned} \mathbf{y}_1 &= [0.35 \quad 0.49 \quad -0.65 \quad -0.65 \quad -0.69 \quad 0.48]^T \\ \mathbf{y}_2 &= [-0.05 \quad -0.06 \quad -0.28 \quad -0.21 \quad 0.13 \quad 0.37]^T \\ \mathbf{y}_3 &= [0.48 \quad 0.50 \quad -0.77 \quad -1.66 \quad -0.76 \quad 0.71]^T. \end{aligned}$$

The element-wise output from the ReLU activation function is then

$$\begin{aligned} f(\mathbf{y}_1) &= [0.35 \quad 0.49 \quad \mathbf{0.00} \quad \mathbf{0.00} \quad \mathbf{0.00} \quad 0.48]^T \\ f(\mathbf{y}_2) &= [\mathbf{0.00} \quad \mathbf{0.00} \quad \mathbf{0.00} \quad \mathbf{0.00} \quad 0.13 \quad 0.37]^T \\ f(\mathbf{y}_3) &= [0.48 \quad 0.50 \quad \mathbf{0.00} \quad \mathbf{0.00} \quad \mathbf{0.00} \quad 0.71]^T, \end{aligned}$$

It is also convenient to consider the indicator matrix that designates the active/deactivated neurons after the ReLU activation, which is given by

$$\mathbf{M}^{\text{ReLU}} = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}^T.$$

Since the ReLU is defined in such a way that it produces a zero output for negative input values, the main problem with the ReLU activation function arises when the input to a neuron has many negative values so that the corresponding zero-output of the ReLU function will leave many neurons without update of their weights (“dying ReLU”). This problem can be avoided using Leaky ReLU, whereby negative values of the input are mapped onto small scaling factors, for example, $f(y_k(n)) = 0.01y_k(n)$, for $y_k(n) < 0$.

- 6) **Stride – Convolution step (down-sampling):** In calculating the convolution, it is common that the convolution filter is shifted along by one step, so that after $y_k(n)$ is obtained at an instant n , the next convolution is calculated at $(n+1)$. However, if the signal is sufficiently dense and slow-varying, for computational reasons we may decide to skip several time instants before the next convolution, $y_k(n)$, is calculated. This operation effectively represents downsampling of the output signal in the convolution calculation, whereby the degree of downsampling is called the *stride* (step). For example, if one time instant (or pixel in both directions) is skipped before the next calculation of the convolution value, then the stride value is equal to two, which corresponds to the down-sampling the convolution output, $y_k(n)$, by a factor of 2. The stride value of four, which would mean that the convolution $y_k(n)$ is calculated at every fourth instant (pixel) of the original signal $x(n)$.

Example 4. Consider the output from the ReLU activation function in Example 3, given by

$$\begin{aligned} f(\mathbf{y}_1) &= [0.35 \quad 0.49 \quad 0.00 \quad \mathbf{0.00} \quad 0.00 \quad 0.48]^T \\ f(\mathbf{y}_2) &= [\mathbf{0.00} \quad 0.00 \quad 0.00 \quad \mathbf{0.00} \quad 0.13 \quad 0.37]^T \\ f(\mathbf{y}_3) &= [0.48 \quad 0.50 \quad 0.00 \quad \mathbf{0.00} \quad 0.00 \quad 0.71]^T. \end{aligned}$$

The output with stride 3 is obtained by down-sampling the outputs $f(\mathbf{y}_k)$ by the factor of 3 to give $\text{Stride}_3\{f(\mathbf{y}_1)\} = [0.35 \quad 0.00]^T$
 $\text{Stride}_3\{f(\mathbf{y}_2)\} = [0.00 \quad 0.00]^T$
 $\text{Stride}_3\{f(\mathbf{y}_3)\} = [0.48 \quad 0.00]^T.$

The indicator matrix which corresponds to upsampling (inserting zeros) from $\text{Stride}_3\{f(\mathbf{y}_k)\}$ to the original solution $f(\mathbf{y}_k)$ is then given by

$$\mathbf{M}^{\text{Stride}_3} = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}^T.$$

- 7) **Pooling:** In order to reduce the possibly excessive size of the data throughput, the output signals at each layer are typically further down-sampled through the so called *pooling* operation, in addition to the stride type of down-sampling scheme described above. A typical pooling operation of $y_k(n)$ in a CNN is the max-pooling which splits the output signal into nonoverlapping segments of P samples and takes the maximum value from each segment (for an image, we split the image into $P \times P$ nonoverlapping segments and take the maximum value from every such segment). The signal at the output of the *max-pooling* step with P segments then becomes

$$o_k^1(n) = \max\{f(y_k(n)), f(y_k(n+1)), \dots, f(y_k(n+P-1))\} = F_1(x(n), w(n))$$

or in a vector form

$$\mathbf{o}_k^1 = F_1(\mathbf{w}_k^1, \mathbf{x}).$$

The max-pooling reduces the size of the representation, and thus helps decrease of the computation requirements and the number of weights in a CNN. Pooling also provides some translation invariance, since it chooses the maximum value among P neighboring samples, regardless of their position. Other forms of pooling include the average-pooling, whereby the output is an average of P neighboring samples.

Example 5. Consider the output from the ReLU activation function in Example 3. Then, the output from the max-pooling operation, with $P = 3$, is obtained from $f(\mathbf{y}_k)$ as

$$\mathbf{o}^1 = \begin{bmatrix} \max\{0.35 & 0.49 & 0.00\} & \max\{0.00 & 0.00 & 0.48\} \\ \max\{0.00 & 0.00 & 0.00\} & \max\{0.00 & 0.13 & 0.37\} \\ \max\{0.48 & 0.50 & 0.00\} & \max\{0.00 & 0.00 & 0.71\} \end{bmatrix}^T = \begin{bmatrix} 0.49 & 0.48 \\ 0.00 & 0.37 \\ 0.50 & 0.71 \end{bmatrix}^T$$

The corresponding indicator matrix for the upsampling from the downsampled \mathbf{o}^1 to the original size of $f(\mathbf{y}_k)$ is given by

$$\mathbf{M}^{MP} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}^T.$$

Notice that the max-pooling with $P = 3$ reduces the size of $f(\mathbf{y}_k)$ from 6 to 2, the same as when employing the stride factor of 3. However, unlike in the stride operation case, in the max-pooling the positions of the selected samples, and the corresponding upsampling matrix, are signal dependent.

- 8) **Flattening:** The one-dimensional signals \mathbf{o}^1 , after pooling, are already in a vector form. These vectors are then concatenated to form the vector with the elements

$$o_F^1((k-1)(N-M+1)+n) = o_k^1(n), \text{ for } k = 1, 2, \dots, K \text{ and } n = 0, 1, \dots, N-M.$$

This vector is of size $K(M-M+1)$ if no max-pooling is performed. If max-pooling with a factor of P is used, the size of the concatenated (flattened) vector \mathbf{o}_F^1 is $K(M-M+1)/P$.

In the case of images, while after the max-pooling the output still remains a two-dimensional object; these images are also rearranged into the vector form and concatenated into one vector.

This process is called the *flattening* operation.

Example 6. Fig. 3 depicts the operation of the first convolution layer of a CNN for an input signal of $N = 32$ samples each, four convolution filters ($K = 4$) with $M = 5$ samples, the ReLU nonlinear activation function $\max\{0, y_k^1 + b_k^1\}$, $k = 1, 2, 3, 4$, and max pooling with factor $P = 2$. At the max-pooling stage, the signal is grouped into segments of two samples and the largest sample then represents the output of this operation. The output from the max-pooling stage is then either used as input to the next convolution layer (in the case of multiple convolutional layers) or it is flattened and fed to the neurons of a common fully connected (FC) neural network. The initial weights of the convolution filter are generated as Gaussian random numbers (common way for CNN initialization).

- 9) **Repeated convolutions:** Notice that before flattening, the convolution steps can be repeated one or more times, involving different sets of filter functions (features). Such repeated convolutions help to find possible hierarchically composed features. The convolutional steps can be repeated with or without the activations and pooling functions, referred to as *convolution-activation-pooling*.

Example 7. The output signals, $o_1^1(n)$, $o_2^1(n)$, $o_3^1(n)$, and $o_4^1(n)$ from the first convolutional layer in Example 6 are used as input to the second convolutional layer of a CNN, as shown in Fig. 4. These signals are processed with $K = 5$ convolutional filters, $w_{1,p}^2(n)$, $w_{2,p}^2(n)$, $w_{3,p}^2(n)$, $w_{4,p}^2(n)$, and $w_{5,p}^2(n)$, each of length $M = 3$ and for $p = 1, 2, 3, 4$. The output of the convolutional filters in the second layer is denoted by $y_1^2(n)$, $y_2^2(n)$, $y_3^2(n)$, $y_4^2(n)$, and $y_5^2(n)$. The ReLU activation function is applied to these signals to produce, $\max\{0, y_k^2(n) + b_k^2\}$, $k = 1, 2, 3, 4, 5$. The max-pooling stage with factor of $P = 2$ is next used to produce the signals $o_1^2(n)$, $o_2^2(n)$, $o_3^2(n)$, $o_4^2(n)$ and $o_5^2(n)$. Finally, the flattened output of the second convolutional layer, denoted by $o_F^2(n)$, is formed to serve as an input to the FC layer.

- 10) **Fully Connected (FC) Layers:** The output of the previous convolutional steps, after flattening, are connected in the form of the flattened data to the standard neural network with fully connected neurons. Neurons in this layer have full connectivity with all neurons in the preceding and following layers, as seen in regular feed-forward neural networks. The

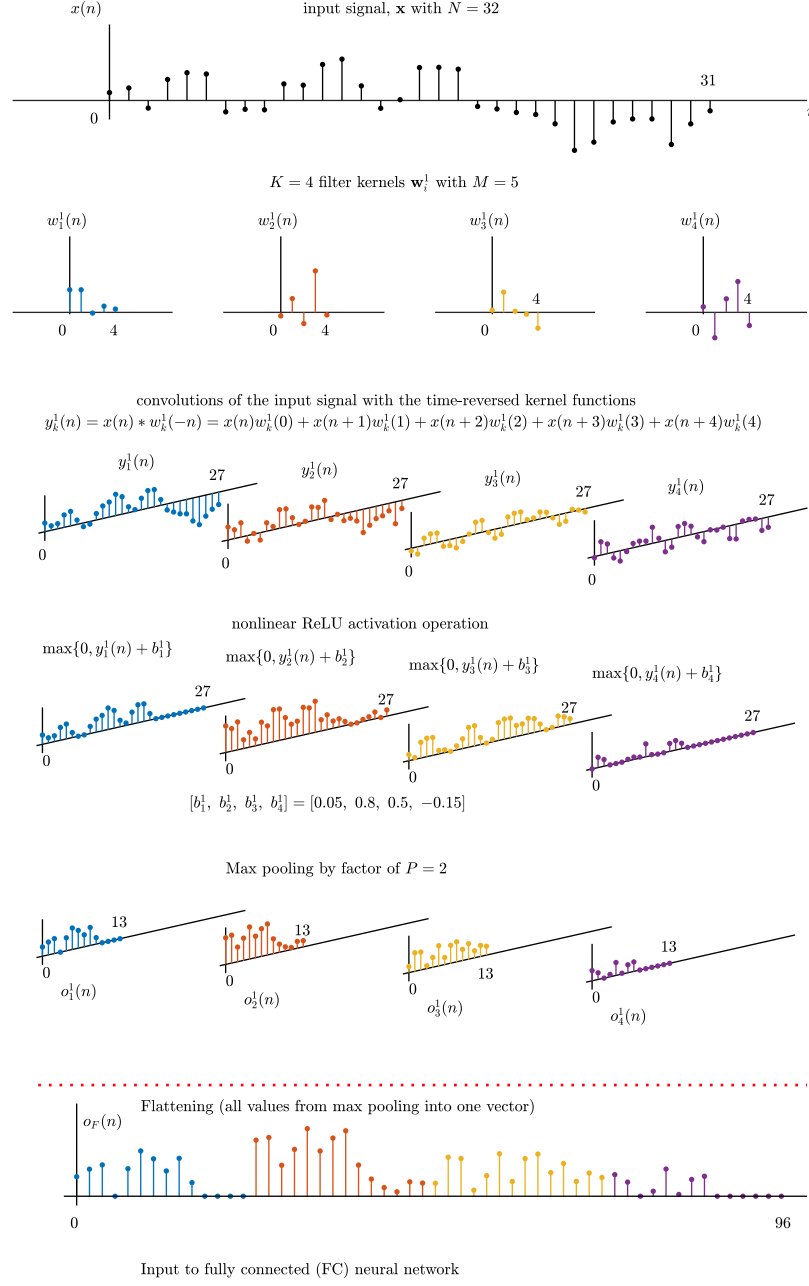


Fig. 3. Illustration of the operation of the first convolutional layer for a CNN with an input signal of $N = 32$ samples each, four convolution filters ($K = 4$) with $M = 5$ samples, the ReLU nonlinear activation function $\max\{0, y_k^1(n) + b_k^1\}$, $k = 1, 2, 3, 4$, and max pooling with the factor $P = 2$. The output from the max pooling operation is used either as input to the next convolution layer, or it is flattened and presented to the neurons of a standard fully connected (FC) neural network layer. The weights of the convolution filter are generated as Gaussian random numbers (common way for the CNN initialization).

FC layers may have a traditional multilayer structure, and are followed by the output layer, which is described in the sequel.

Example 8. An input signal, $x(n)$, with $N = 32$ samples, serves as input to the one-dimensional convolutional layer in a CNN, as shown in Fig. 3 and Fig. 4. The signal is processed with $K = 4$ convolutional filters, $w_1^1(n)$, $w_2^1(n)$, $w_3^1(n)$, and $w_4^1(n)$, each of length $M = 3$. The output of these convolutional filters is given by $y_1^1(n)$, $y_2^1(n)$, $y_3^1(n)$, and $y_4^1(n)$. The ReLU activation function is applied to these signals to produce, $\max\{0, y_1^1(n) + b_1^1\}$, $\max\{0, y_2^1(n) + b_2^1\}$, $\max\{0, y_3^1(n) + b_3^1\}$, and $\max\{0, y_4^1(n) + b_4^1\}$. The max-pooling with factor $P = 2$ yields the signals $o_1^1(n)$, $o_2^1(n)$, $o_3^1(n)$, and $o_4^1(n)$. Finally, the flattened output of this layer is formed, and denoted by $o_F^1(n)$.

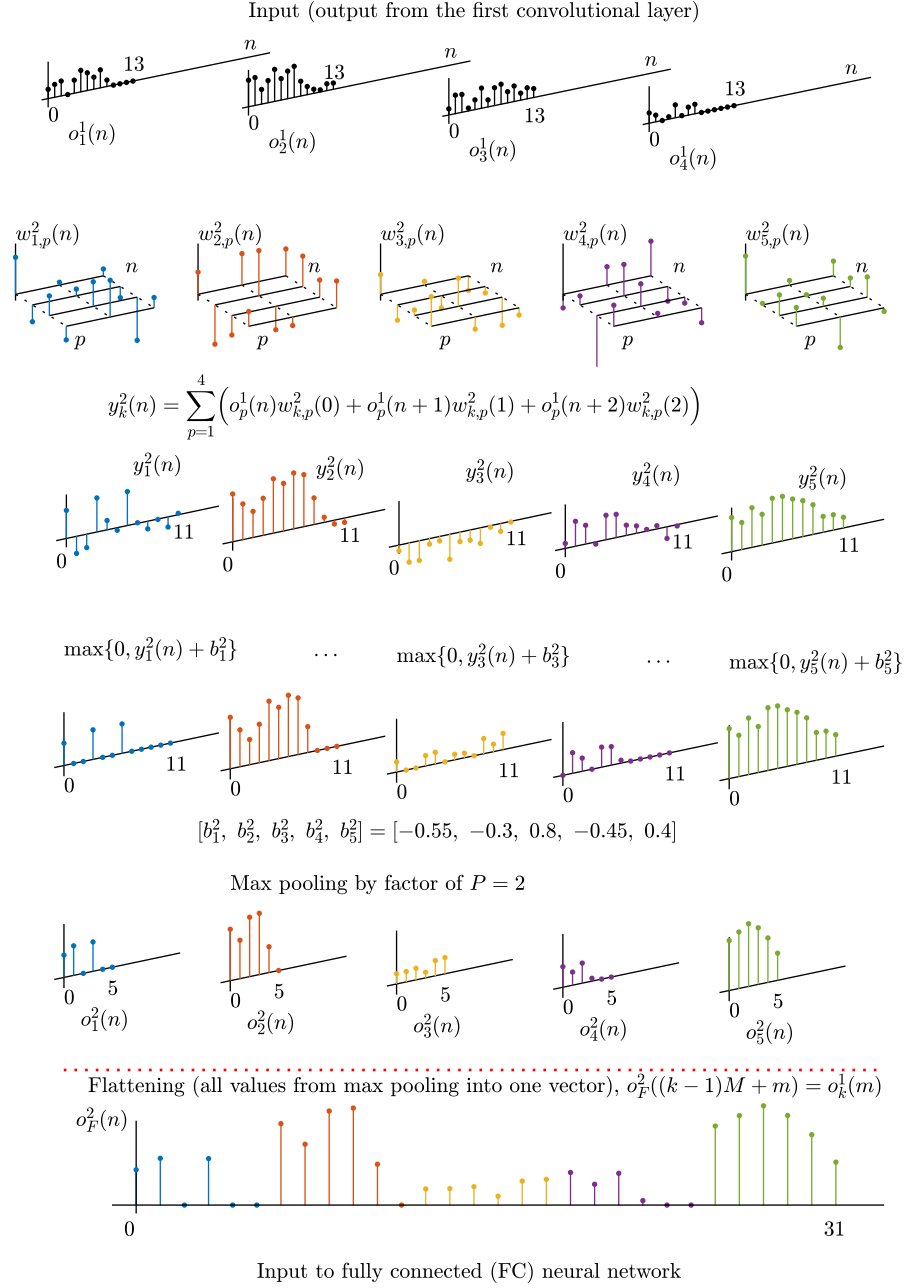
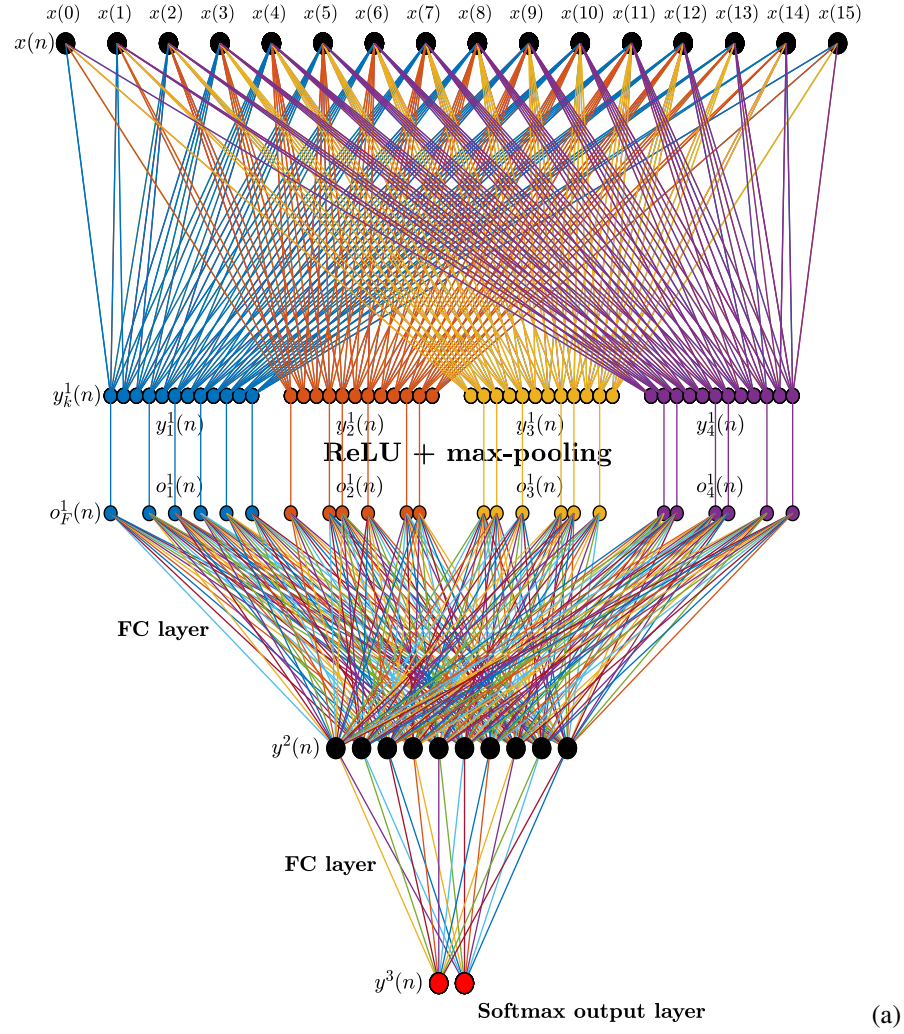


Fig. 4. Operation of the second convolution layer for a CNN which uses the output from the first layer as its input. Five convolution filters ($K = 5$) with $M = 3$ samples were used, the ReLU nonlinear activation function, $\max\{0, y_i^2 + b_i^2\}$, and max pooling with factor $P = 2$, whereby the signal from the previous step is grouped into segments of two samples with the largest sample serving as the output. The signal from the max pooling is used either as an input to the next convolution layer or it is flattened (if only one convolutional layer is used) and fed to fully connected (FC) of a standard neural network. The weights of the convolution filter are generated as Gaussian random numbers (common way for the CNN initialization).

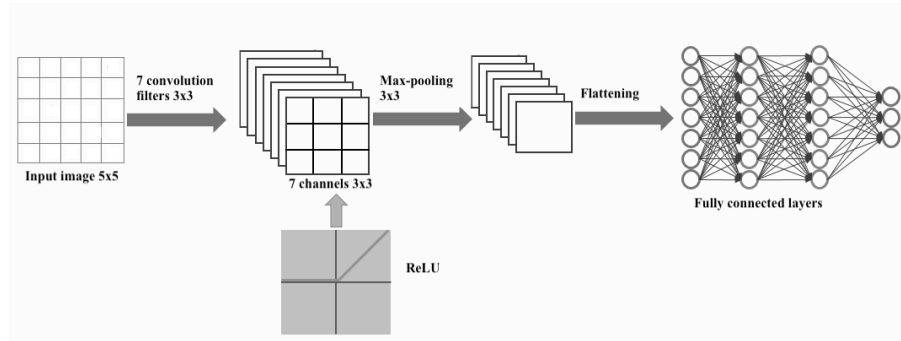
Example 9. To illustrate the sheer number of parameters required in one successful example of a CNN we quote the authors of *AlexNet*:

"We trained a large, deep convolutional neural network to classify the 1.2 million high-resolution images in the ImageNet LSVRC-2010 contest into the 1000 different classes. The neural network, which has 60 million parameters and 650,000 neurons, consists of five convolutional layers, some of which are followed by max-pooling layers, and three fully connected layers with a final 1000-way softmax.

The first convolutional layer filters the 224x224x3 input image with 96 kernels of size 11x11x3 with a stride of 4 pixels (this is the distance between the receptive field centers of neighboring neurons in a kernel map). The second convolutional layer takes as input the (response-normalized and pooled) output of the first convolutional layer and filters it with 256 kernels of size 5x5x48. The third, fourth, and fifth convolutional layers are connected to one another without any intervening pooling or normalization layers. The third convolutional layer has 384 kernels of size 3x3x256 connected to the (normalized, pooled) outputs of the second convolutional layer. The fourth convolutional layer has 384 kernels of size 3x3x192, and the fifth convolutional layer has 256 kernels of size 3x3x192. The



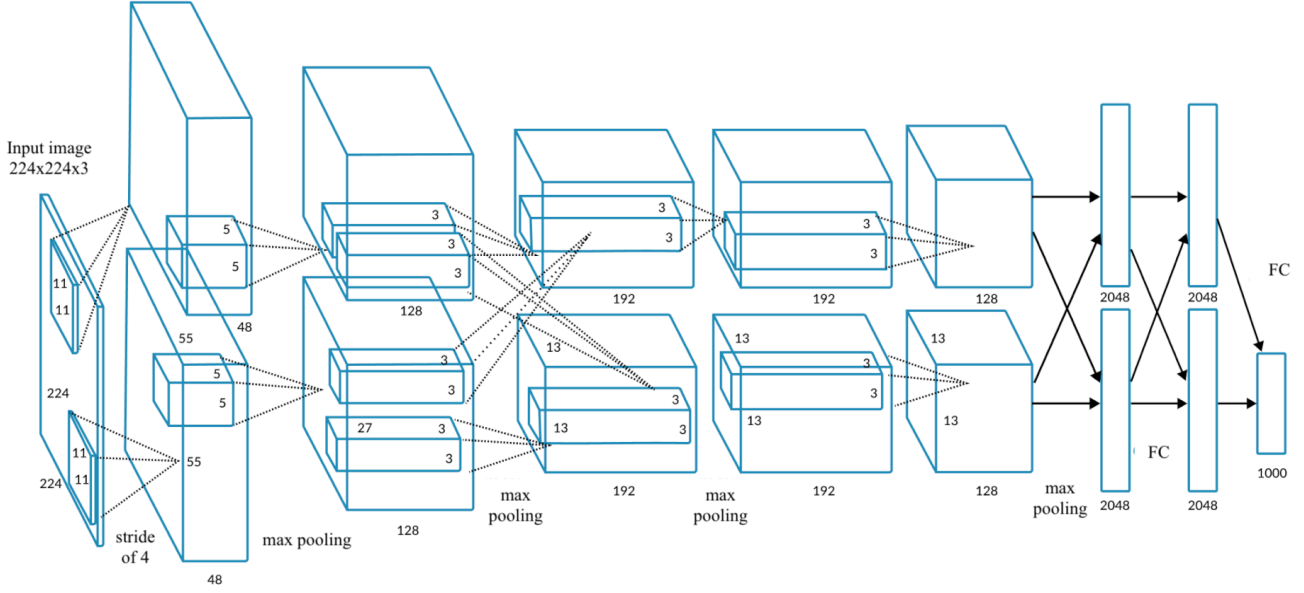
(a)



(b)

Fig. 5. An exemplar of architecture of CNNs for signals and images. (a) Illustration of a CNN for signals, with one convolution layer and two FC layers, with two neurons at the output (softmax) layer. (b) Illustration of a CNN for images with one convolution layer and two FC layers, with three neurons at the output (softmax) layer.

fully-connected layers have 4096 neurons each.” A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet Classification with Deep Convolutional Neural Networks”, *Communications of the ACM*, 60 (6): 84–90, May 2017.



AlexNet architecture [19]:

[224x224x3] INPUT COLOR IMAGES

CONV1: 96 11x11 filters at stride 4, pad 0 $\rightarrow 2 \times [55 \times 55 \times 48] = [55 \times 55 \times 96]$

MAX POOL1: 3x3 filters at stride 2 $\rightarrow [27 \times 27 \times 96]$

CONV2: 256 5x5 filters at stride 1, pad 2 $\rightarrow [27 \times 27 \times 96]$

MAX POOL2: 3x3 filters at stride 2 $\rightarrow [13 \times 13 \times 256]$

CONV3: 384 3x3 filters at stride 1, pad 1 $\rightarrow [13 \times 13 \times 256]$

CONV4: 384 3x3 filters at stride 1, pad 1 $\rightarrow [13 \times 13 \times 384]$

CONV5: 256 3x3 filters at stride 1, pad 1 $\rightarrow [13 \times 13 \times 256]$

MAX POOL3: 3x3 filters at stride 2 $\rightarrow [6 \times 6 \times 256]$

[4096] FC6: 4096 neurons

[4096] FC7: 4096 neurons

[1000] FC8: 1000 neurons (class scores)

Fig. 6. AlexNet was designed by SuperVision group (2012) [19]. This deep CNN is used to classify the 1.2 million high-resolution images in the ImageNet LSVRC-2010 contest into the 1000 different classes. The network has 60 million parameters and 650,000 neurons, consists of five convolutional layers, some of which are followed by max-pooling layers, and three fully connected layers with a final 1000-way softmax.

IV. UPDATING CONVOLUTION WEIGHTS: BACK-PROPAGATION

The initial parameters (weights) of a CNN are typically updated in a supervised way through a gradient-based learning process known as the back-propagation (BP) algorithm. For each iteration of the BP, the gradient magnitude (or sensitivity) of each network parameter (such as the weights of the convolutional or the fully-connected layers) is computed. These parameter sensitivities are then used to iteratively update the CNN parameters until a certain stopping criterion is met or the training data set is exhausted.

1) *Initialization*: Unlike standard adaptive systems, where the initial weight values are typically set to zero, in neural networks the initial values of the weights are typically assumed as random (and different) values for each channel and layer. Since the weights $w_k(m)$ multiply, in general, N_{in} input signal values (at the considered input neurons of the layer), the only requirement is that the choice of the initial weights preserves the expected energy of the output for the considered layers. This is achieved, for example, if the initial weights are *Gaussian distributed*, with

$$w_k(m) \sim \sqrt{\frac{2}{N_{in}}} \mathcal{N}(0,1).$$

The factor of 2 is used since the ReLU activation function will remove negative output values, which accounts for a half of the expected energy.

Another possibility is to use *uniformly* distributed initial wights, $w_k(m)$, whereby the sum of N_{in} initial weights, $\sum_{m=0}^{N_{in}} w_k(m)$, produces a unit variance. Such uniformly distributed weights are defined by the interval

$$w_k(m) \sim \left[-\sqrt{\frac{6}{N_{in}}}, \sqrt{\frac{6}{N_{in}}} \right].$$

The variance of this random variable is $\text{Var}\{w_k(m)\} = \frac{6}{N_{in}} \frac{1}{3}$. The variance of a sum of N_{in} values, divided by 2, to take into account the ReLU, produces unit variance. Such initial values are called the He initial values.

If the previous values of the initial weights are additionally reduced, taking into account the number of output neurons for the considered layer, N_{out} , then the Xavier initial values are obtained

$$w_k(m) \sim \sqrt{\frac{2}{N_{in} + N_{out}}} \mathcal{N}(0, 1)$$

or

$$w_k(m) \sim \left[-\sqrt{\frac{6}{N_{in} + N_{out}}}, \sqrt{\frac{6}{N_{in} + N_{out}}} \right].$$

2) *Back-propagation in a two-layer CNN*: Consider first the weight update in the simplest CNN which consists of two layers, a convolutional layer and a fully connected output layer.

Convolutional layer. For the input $\mathbf{x} = [x(0), x(1), \dots, x(N-1)]^T$, the output signal of the convolutional layer of the CNN, with K filters of the width M , is given by

$$y_k^1(n) = w_k^1(0)x(n) + w_k^1(1)x(n+1) + \dots + w_k^1(M-1)x(n+M-1) = \sum_{m=0}^{M-1} w_k^1(m)x(n+m), \quad (5)$$

for the channels $k = 1, 2, \dots, K$, as shown in Fig. 3. The overall output of the convolution layer is then obtained after the bias term is included and upon the application of the ReLU activation function, to yield

$$o_k^1(n) = f(y_k^1(n) + b_k^1). \quad (6)$$

For simplicity, we shall first assume that no max-pooling or any other down-sampling is performed.

The output from the convolutional layer is then stacked into a vector of length $K(N-M+1)$, which serves as input to the fully connected layer with S outputs.

Each of $K(N-M+1)$ nodes of the output of the convolutional layer, with the signal samples,

$$[o_1^1(0), \dots, o_1^1(N-M), o_2^1(0), \dots, o_2^1(N-M), \dots, o_K^1(0), \dots, o_K^1(N-M)]^T$$

is connected to each of the S nodes of the fully connected output layer to produce the overall CNN output of the form

$$\begin{aligned} y_k^2 &= w_k^2(0)o_1^1(0) + w_k^2(1)o_1^1(1) + \dots + w_k^2(N-M)o_1^1(N-M) \\ &\quad + w_k^2(N-M+1)o_2^1(0) + \dots + w_k^2(2(N-M+1)-1)o_2^1(N-M) \\ &\quad \vdots \\ &\quad + w_k^2((K-1)(N-M+1))o_K^1(0) + \dots + w_k^2(K(N-M+1)-1)o_K^1(N-M), \end{aligned} \quad (7)$$

for $k = 1, 2, \dots, S$. Note that the number of weights in the k th FC layer is $SK(N-M+1)$.

A commonly used *loss function* in the minimization is the mean square error (MSE) between the network prediction and the true lable, given by

$$\mathcal{L} = \frac{1}{2} \sum_{k=1}^S (y_k^2 - t_k)^2, \quad (8)$$

where t_k is the desired or target output (also called a teaching signal).

Training process. To define the gradient descent relations for the update of all previous weights (within the convolutional layer and the fully connected layer) in the training process, consider first the convolutional layer, in (5)-(6), to give the gradient weight update in the form

$$w_k^1(m)_{new} = w_k^1(m)_{old} - \alpha \frac{\partial \mathcal{L}}{\partial w_k^1(m)} \big|_{w_k^1(m)=w_k^1(m)_{old}}. \quad (9)$$

The element-wise gradient values are then calculated

$$\frac{\partial \mathcal{L}}{\partial w_k^1(m)} = \sum_n \frac{\partial \mathcal{L}}{\partial y_k^1(n)} \frac{\partial y_k^1(n)}{\partial w_k^1(m)} = \sum_n \frac{\partial \mathcal{L}}{\partial y_k^1(n)} x(n+m) = \frac{\partial \mathcal{L}}{\partial y_k^1(m)} *_c x(m), \quad (10)$$

where (5) is used for the calculation¹ of $\partial y_k^1(n)/\partial w_k^1(m)$.

Next, we need to calculate the so called *delta error* function $\partial \mathcal{L}/\partial y_k^1(m) = \Delta_k^1(m)$, which can be written as

$$\begin{aligned} \Delta_k^1(m) &= \frac{\partial \mathcal{L}}{\partial y_k^1(m)} = \sum_p \frac{\partial \mathcal{L}}{\partial y_p^2} \frac{\partial y_p^2}{\partial y_k^1(m)} = \sum_p \frac{\partial \mathcal{L}}{\partial y_p^2} \frac{\partial y_p^2}{\partial o_k^1(m)} \frac{\partial o_k^1(m)}{\partial y_k^1(m)} \\ &= \sum_p \Delta_p^2 w_p^2((k-1)M+m) u(y_k^1(m)) \end{aligned} \quad (11)$$

where the relation in (7) is used for the calculation of $\partial y_p^2/\partial o_k^1(m) = w_p^2((k-1)M+m)$ and

$$\Delta_p^2 = \frac{\partial \mathcal{L}}{\partial y_p^2} = y_p^2 - t_p$$

is the error in the final stage.

The relation in (11) back-propagates the error from layer 2, denoted by Δ_p^2 , to layer 1, to yield a portion of the overall error attributed to neuron k of layer 1, $\Delta_k^1(m)$. We can now calculate $\partial \mathcal{L}/\partial y_k^1(m) = \Delta_k^1(m)$ and the gradient for the update in (9).

The bias terms are updated in the same way

$$b_{k,new}^1 = b_{k,old}^1 - \alpha \frac{\partial \mathcal{L}}{\partial b_k^1} \big|_{b_k^1 = b_{k,old}^1} \quad (12)$$

and

$$\frac{\partial \mathcal{L}}{\partial b_k^1} = \sum_n \frac{\partial \mathcal{L}}{\partial y_k^1(n)} \frac{\partial y_k^1(n)}{\partial b_k^1} = \sum_n \frac{\partial \mathcal{L}}{\partial y_k^1(n)} = \sum_n \Delta_k^1(n) \quad (13)$$

If the max-pooling operation is used, then the output $y_k(n)$ is used only for some $n \in \mathbb{M}_k$, and the gradient update is adjusted accordingly, as

$$\frac{\partial \mathcal{L}}{\partial w_k^1(m)} = \sum_{n \in \mathbb{M}_k} \frac{\partial \mathcal{L}}{\partial y_k^1(n)} \frac{\partial y_k^1(n)}{\partial w_k^1(m)} = \sum_{n \in \mathbb{M}_k} \frac{\partial \mathcal{L}}{\partial y_k^1(n)} x(n+m). \quad (14)$$

Notice that within the max-pooling, the convolution values used at $n \in \mathbb{M}_k$ may change at each update step. If the stride is also used, then the values of $y_k(n)$ are calculated according to a defined stride step. For example, with the stride value of 2, the convolutions are always calculated at $y_k(0), y_k(2), \dots, y_k(N-2)$.

Fully Connected (FC) layer. The input to the FC layer represents the flattened output from the convolutional layer, given by

$$o_F^1((k-1)(N-M+1)+m) = o_k^1(n).$$

The indices n in $o_F^1(n)$ range from 0 to $K(N-M+1)-1$. Notice that relation (7) could be equally written as

$$y_k^2 = \sum_{n=0}^{K(N-M+1)-1} w_k^2(n) o_F^1(n).$$

The update of the fully connected layer weights, $w_k^2(n)$, is performed in the same way, using

$$w_k^2(m)_{new} = w_k^2(m)_{old} - \alpha \frac{\partial \mathcal{L}}{\partial w_k^2(m)} \big|_{w_k^2(m) = w_k^2(m)_{old}}, \quad (15)$$

with the gradient elements in the form

$$\frac{\partial \mathcal{L}}{\partial w_k^2(m)} = \frac{\partial \mathcal{L}}{\partial y_k^2} \frac{\partial y_k^2}{\partial w_k^2(m)} = (y_k^2 - t_k) o_F^1(m).$$

¹Here, we have also used the property of an implicit function derivative, given by

$$\begin{aligned} \frac{\partial F(u(x,y,z), v(x,y,z), w(x,y,z))}{\partial x} &= \frac{\partial F(u(x,y,z), v(x,y,z), w(x,y,z))}{\partial u} \frac{\partial u}{\partial x} \\ &+ \frac{\partial F(u(x,y,z), v(x,y,z), w(x,y,z))}{\partial v} \frac{\partial v}{\partial x} + \frac{\partial F(u(x,y,z), v(x,y,z), w(x,y,z))}{\partial w} \frac{\partial w}{\partial x}. \end{aligned}$$

Notice that this relation is a *special case* of (14), for the CNN with $y_k^2(n) = y_k^2(0) = y_k^2$, that is, when the summation over n in (14) reduces to one term only for $n = 0$.

If a nonlinear activation function is used at the output, then the factor of $f'(y_k^2)$ should multiply the right hand side of $\partial\mathcal{L}/\partial w_k^2(m)$.

3) *Softmax Output Layer*: In some applications, the output layer gives the probabilities for the decision when classifying of the analyzed data. The output therefore represents a list of probabilities for different possible labels (basins of attraction) associated with the analyzed signal or image (for example, dog, cat, bird in the image), whereby the label that receives the highest probability is the classification decision. In the error calculation, the desired (target) output then assumed the value $t_k = 1$ for one value $k = k_0$ (in training process we know what signal/image is analyzed by the CNN) and $t_k = 0$ for other values of k .

Since the output, y_k^L , from the last L th layer (overall output), may assume various positive and negative real values, we need to map the output y_k^L into probability-like values, using a function of the form

$$P_k = \frac{e^{y_k^L}}{\sum_{i=1}^S e^{y_i^L}}. \quad (16)$$

called the softmax. Obviously, $0 \leq P_k \leq 1$ and $\sum_{k=1}^S P_k = 1$.

When the softmax is used as the output mapping, the loss function is modified accordingly, from the mean square error to the cross-entropy form, given by

$$\mathcal{L} = - \sum_{k=1}^S t_k \ln(P_k).$$

This cross-entropy is very large if there is a t_k close to 1, but the corresponding output probability P_k is small, meaning that a big change in the weights should be performed. The cross-entropy, \mathcal{L} , is small only when for $t_{k_0} = 1$ at a specific k_0 , and the value of corresponding P_{k_0} is close to 1.

We can easily show that the delta error function in the output layer is of the form

$$\Delta_k^L = \frac{\partial\mathcal{L}}{\partial y_k^L} = \sum_{i=1}^S \frac{\partial\mathcal{L}}{\partial P_i} \frac{\partial P_i}{\partial y_k^L} = \sum_{i=1}^S \left(\frac{t_i}{P_i} P_i P_k \right) - \frac{t_k}{P_k} P_k = P_k - t_k$$

since from (16) it follows that $\partial P_i / \partial y_k^L = -P_i P_k$ if $i \neq k$ and $\partial P_i / \partial y_k^L = P_i(1 - P_k) = -P_i P_k + P_k$ if $i = k$, while $\sum_{i=1}^S t_i = 1$.

Therefore, as expected, there is no weight correction if $t_k = P_k$, while, as desired, all the previous (and next) relations regarding the back-propagation also hold in this case.

4) *Back-Propagation in a Multi-Layer CNN*: After the back-propagation is illustrated for a simple two-layer network example, we can now generalize the back-propagation relations to a multi-layer CNN. The output in the layer l , $l = 1, 2, \dots, L$, of a general CNN without max-pooling, is defined by

$$\mathbf{y}_k^l = \sum_p \mathbf{o}_p^{l-1} * \mathbf{w}_{k,p}^l + b_k^l,$$

where \mathbf{o}^{l-1} is the output of the layer $(l-1)$, as shown in Fig. 4. The element-wise form of this output is given by

$$y_k^l(n) = \sum_{p=1}^K \sum_{m=0}^{M-1} \left(o_p^{l-1}(n+m) w_{k,p}^l(m) \right) + b_k^l. \quad (17)$$

Notice that the input to the layer input is equal to the input signal, $\mathbf{o}^0 = \mathbf{x}$. For any other layer we have

$$\mathbf{o}^{l-1} = f(\mathbf{y}_k^{(l-1)}),$$

where $f(x)$ is the nonlinear activation function (commonly ReLU in the CNN).

Next, we specify those derivatives in (17) that will be used in the update of neural network weights

$$\begin{aligned} \frac{\partial y_k^l(n)}{\partial w_{k,p}^l(m)} &= o_p^{(l-1)}(n+m) \\ \frac{\partial y_k^{l+1}(n-\mu)}{\partial o_q^l(n)} &= \frac{\partial}{\partial o_q^l(n)} \left(\sum_{p=1}^K \sum_{m=0}^{M-1} \left(o_p^l(n-\mu+m) w_{k,p}^{l+1}(m) \right) + b_k^{l+1} \right) = w_{k,q}^{l+1}(\mu) \\ \frac{\partial o_p^l(n)}{\partial y_k^l(n)} &= f'(y_k^l(n)) = u(y_k^l(n)), \end{aligned}$$

where $u(x)$ is the unit step function.

Gradients of weight update. The weights should be changed according to the gradient descent direction of the loss function, \mathcal{L} , that is

$$w_{k,p}^l(m)_{new} = w_{k,p}^l(m)_{old} - \alpha \frac{\partial \mathcal{L}}{\partial w_{k,p}^l(m)} \Big|_{w_{k,p}^l(m)=w_{k,p}^l(m)_{old}}. \quad (18)$$

- For the convolutional layer, the derivative of the cost function with respect to $w_{k,p}^l(m)$, using the previously stated derivatives, becomes

$$\frac{\partial \mathcal{L}}{\partial w_{k,p}^l(m)} = \sum_n \frac{\partial \mathcal{L}}{\partial y_k^l(n)} \frac{\partial y_k^l(n)}{\partial w_{k,p}^l(m)} = \sum_n \frac{\partial \mathcal{L}}{\partial y_k^l(n)} o_p^{(l-1)}(n+m) = \frac{\partial \mathcal{L}}{\partial y_k^l(m)} *_c o_p^{(l-1)}(m).$$

- For the standard, fully connected layer, according to (4), the following holds

$$\frac{\partial \mathcal{L}}{\partial w_k^l(m)} = \frac{\partial \mathcal{L}}{\partial y_k^l(0)} \frac{\partial y_k^l(0)}{\partial w_k^l(m)} = \frac{\partial \mathcal{L}}{\partial y_k^l} \frac{\partial y_k^l}{\partial w_k^l(m)} = \frac{\partial \mathcal{L}}{\partial y_k^l} o_k^{(l-1)}(m).$$

5) *Delta error back-propagation:* In the CNN jargon, the derivative $\partial \mathcal{L} / \partial y_k^l(m) = \Delta_k^l(m)$ is called the *delta error*. For an arbitrary layer l , it should be related to the error function in the last (output) layer, $\Delta_k^L = P_k - t_k$. By using the composition of derivatives, we can relate the delta error in the l th layer with that in the next, $(l+1)$ th, layer, and then propagate this relation iteratively to the output layer. This can be written as

$$\begin{aligned} \Delta_k^l(n) &= \frac{\partial \mathcal{L}}{\partial y_k^l(n)} = \sum_m \sum_p \frac{\partial \mathcal{L}}{\partial y_p^{l+1}(n-m)} \frac{\partial y_p^{l+1}(n-m)}{\partial y_k^l(n)} \\ &= \sum_m \sum_p \frac{\partial \mathcal{L}}{\partial y_p^{l+1}(n-m)} \frac{\partial y_p^{l+1}(n-m)}{\partial o_k^l(n)} \frac{\partial o_k^l(n)}{\partial y_k^l(n)} = \sum_m \sum_p \frac{\partial \mathcal{L}}{\partial y_p^{l+1}(n-m)} w_{p,k}^{l+1}(m) u(y_k^l(n)). \end{aligned}$$

Back-propagation of the delta error. From the above, the recursive back-propagation relation for the delta error calculation in the convolutional layer is given by

$$\Delta_k^l(n) = u(y_k^l(n)) \sum_p \left(\sum_m \frac{\partial \mathcal{L}}{\partial y_p^{l+1}(n-m)} w_{p,k}^{l+1}(m) \right) = u(y_k^l(n)) \sum_p \left(\Delta_p^{l+1}(n) * w_{k,p}^{l+1}(n) \right),$$

with the final value (the initial value for the back-propagation) for the mean square error

$$\Delta_k^L = \frac{\partial \mathcal{L}}{\partial y_k^L} = \frac{1}{\partial y_k^L} \left(\frac{1}{2} \sum_p (y_p^L - t_p)^2 \right) = y_k^L - t_k,$$

while for the Softmax layer we have

$$\Delta_k^L = P_k - t_k$$

for the Softmax layer.

For a fully connected layer, in the standard neural network, we obtain

$$\Delta_k^l = \frac{\partial \mathcal{L}}{\partial y_k^l} = \sum_p \frac{\partial \mathcal{L}}{\partial y_p^{l+1}} \frac{\partial y_p^{l+1}}{\partial y_k^l} = \sum_p \frac{\partial \mathcal{L}}{\partial y_p^{l+1}} \frac{\partial y_p^{l+1}}{\partial o^l(k)} \frac{\partial o^l(k)}{\partial y_k^l} = \sum_p \Delta_p^{l+1} w_{p,k}^{l+1}(k) u(y_k^l).$$

Bias update. The bias propagation obeys similar rules, and is given by

$$\frac{\partial \mathcal{L}}{\partial b_k^l} = \sum_n \frac{\partial \mathcal{L}}{\partial y_k^l(n)} \frac{\partial y_k^l(n)}{\partial b_k^l} = \sum_n \frac{\partial \mathcal{L}}{\partial y_k^l(n)} = \sum_n \Delta_k^l(n),$$

with

$$b_{k,new}^l = b_{k,old}^l - \alpha \frac{\partial \mathcal{L}}{\partial b_k^l} \Big|_{b_k^l=b_{k,old}^l} \quad (19)$$

For the FC layers, the bias update is performed according to

$$\frac{\partial \mathcal{L}}{\partial b_k^l} = \frac{\partial \mathcal{L}}{\partial y_k^l} \frac{\partial y_k^l}{\partial b_k^l} = \frac{\partial \mathcal{L}}{\partial y_k^l} = \Delta_k^l,$$

with the same update relation as in (19).

Example 10. This example of a two-layer neural network (one convolutional layer and one fully connected layer) illustrates the back-propagation operation in a step-by-step manner. The considered input signal has $N = 8$ samples, which may contain either a variant of the triangular shape pattern, $\text{feature}_1 = [-0.5, 1, -0.5] + v(n)$, ($\mathbf{t} = [0, 1]^T$) or a variant of rectangular three-sample $\text{feature}_2 = [1, 1, 1] + v(n)$, $\mathbf{t} = [1, 0]^T$, where $v(n)$ is random uniform noise whose values lie in the region 0 to 0.3, that introduces deviations in the feature forms. The signal is embedded in additive random Gaussian noise with standard deviation of 0.05, and then normalized to unit energy, as shown in Fig. 7(a). Convolutional filters of $M = 3$ samples are used to produce $K = 3$ channels at the convolutional layer. The Softmax is used at the output of the FC layer, with two values that correspond to the two patterns in the target signal, \mathbf{t} . The network was trained using 200 random signal realizations over 10 epochs (presented 10 times to the network). After the training, the network was tested on 100 new random signal realizations.

Forward calculation: From the input signal to the output	
• Input signal, \mathbf{x}, of length $N = 8$, $\mathbf{x} = [-0.18 \quad -0.28 \quad -0.23 \quad -0.32 \quad 0.45 \quad 0.45 \quad 0.45 \quad -0.35]^T$. The target signal was $\mathbf{t} = [1 \quad 0]^T$, since feature_2 was present in the input.	
• Weight initialization: Random $w_k^1(m) \sim \mathcal{N}(0,1)\sqrt{2/3}$, $M = 3$, for $K = 3$ channels: $\mathbf{w}_1^1 = [-0.07 \quad -0.01 \quad -1.47]^T$, $\mathbf{w}_2^1 = [0.44 \quad 0.14 \quad -0.30]^T$, $\mathbf{w}_3^1 = [1.15 \quad -1.01 \quad -1.83]^T$.	
• Convolutions: $\mathbf{y}_k = \mathbf{x} * \mathbf{w}_k^1 + b_k$, $k = 1, 2, 3$ with the initial bias values $b_1 = 0$, $b_2 = 0$, and $b_3 = 0$. $\mathbf{y}_1 = [0.35 \quad 0.49 \quad -0.65 \quad -0.65 \quad -0.69 \quad 0.48]^T$, $\mathbf{y}_2 = [-0.05 \quad -0.06 \quad -0.28 \quad -0.21 \quad 0.13 \quad 0.37]^T$, $\mathbf{y}_3 = [0.48 \quad 0.50 \quad -0.77 \quad -1.66 \quad -0.76 \quad 0.71]^T$.	
• Nonlinear activation function: ReLU activation function, $\mathbf{F}_k = f(\mathbf{y}_k^1) = \max\{0, \mathbf{y}_k^1\}$, was used, to give $f(\mathbf{y}_1) = [0.35 \quad 0.49 \quad \mathbf{0.00} \quad \mathbf{0.00} \quad \mathbf{0.00} \quad 0.48]^T$, $f(\mathbf{y}_2) = [\mathbf{0.00} \quad \mathbf{0.00} \quad \mathbf{0.00} \quad \mathbf{0.00} \quad 0.13 \quad 0.37]^T$, $f(\mathbf{y}_3) = [0.48 \quad 0.50 \quad \mathbf{0.00} \quad \mathbf{0.00} \quad \mathbf{0.00} \quad 0.71]^T$.	
• Max-pooling: This yields the output $o_k^1(m) = \max\{F_k(mP), \dots, F_k(mP + P - 1)\}$, with $P = 3$, $\mathbf{o}^1 = \begin{bmatrix} \max\{0.35 & 0.49 & \mathbf{0.00}\} & \max\{\mathbf{0.00} & \mathbf{0.00} & 0.48\} \\ \max\{\mathbf{0.00} & \mathbf{0.00} & \mathbf{0.00}\} & \max\{\mathbf{0.00} & 0.13 & 0.37\} \\ \max\{0.48 & 0.50 & \mathbf{0.00}\} & \max\{\mathbf{0.00} & \mathbf{0.00} & 0.71\} \end{bmatrix}^T = \begin{bmatrix} 0.49 & 0.48 \\ \mathbf{0.00} & 0.37 \\ 0.50 & 0.71 \end{bmatrix}^T$, The indicator matrix of chosen value from ReLU, \mathbf{M}^{ReLU} , and max-pooling, \mathbf{M}^{MP} , $\mathbf{M}^{\text{ReLU}} = \begin{bmatrix} \mathbf{1} & \mathbf{1} & 0 & 0 & 0 & \mathbf{1} \\ \mathbf{0} & 0 & 0 & 0 & \mathbf{1} & \mathbf{1} \\ \mathbf{1} & \mathbf{1} & 0 & 0 & 0 & \mathbf{1} \end{bmatrix}^T$ and $\mathbf{M}^{\text{MP}} = \begin{bmatrix} 0 & \mathbf{1} & 0 & 0 & 0 & \mathbf{1} \\ \mathbf{1} & 0 & 0 & 0 & 0 & \mathbf{1} \\ 0 & \mathbf{1} & 0 & 0 & 0 & \mathbf{1} \end{bmatrix}^T$, will be used to reposition the gradient update calculated with the downsampled, \mathbf{o}^1 , to the proper, \mathbf{y}^1 , positions, taking into account possible zeroing by the ReLU, survived from the max-pooling.	
• Flattening: $N_F = (N - M + 1)P = 2$, $o_F^1((k - 1)N_F + m) = o_k^1(m)$, $k = 1, 2, 3$, $m = 0, 1$. $\mathbf{o}_F^1 = [0.49 \quad 0.48 \quad 0.00 \quad 0.37 \quad 0.50 \quad 0.71]^T$,	
• Weight initialization: For the FC layer, random $w_k^2(n) \sim \mathcal{N}(0,1)\sqrt{2/6}$ $\mathbf{w}^2 = \begin{bmatrix} -0.47 & -0.06 & -0.05 & -0.81 & 0.62 & -0.18 \\ 0.02 & 0.12 & -0.15 & -0.07 & -0.87 & -0.53 \end{bmatrix}^T$,	
• Output: From the FC layer, $y_k^2 = \sum_{n=0}^5 o_F^1(n)w_k^2(n)$. $\mathbf{y}^2 = (\mathbf{w}^2)^T \mathbf{o}_F^1 = [-0.38 \quad -0.77]^T$.	
• Softmax: With $S = 2$, $P_k = e^{y_k^2} / (e^{y_1^2} + e^{y_2^2})$, $k = 1, 2$, we get $\mathbf{P} = [0.60 \quad 0.40]^T$, $\Delta^2 = \mathbf{P} - \mathbf{t} = [-0.40 \quad 0.40]^T$, $\Delta_k^2 = P_k - t_k$	

Back-propagation: Delta error, gradient, weight update	
<ul style="list-style-type: none"> Gradient in the FC layer, $g_k^2(m) = \Delta_k^2 o_F^1(m)$, $\mathbf{g}^2 = \mathbf{o}_F^1 (\Delta^2)^T = \begin{bmatrix} -0.12 & -0.12 & -0.00 & -0.09 & -0.12 & -0.17 \\ 0.12 & 0.12 & 0.00 & 0.09 & 0.12 & 0.17 \end{bmatrix}^T$, 	
<ul style="list-style-type: none"> Weight update in the FC layer, $w_k^2(m) \leftarrow w_k^2(m) + 0.1 g_k^2(m)$. $\mathbf{w}^2 = \mathbf{w}^2 - 0.1 \mathbf{g}^2 = \begin{bmatrix} -0.45 & -0.04 & -0.05 & -0.79 & 0.64 & -0.15 \\ 0.00 & 0.10 & -0.15 & -0.08 & -0.89 & -0.56 \end{bmatrix}^T$, 	
<ul style="list-style-type: none"> Delta error back-propagation in the convolutional layer, $\Delta_k^1(m) = \sum_p \Delta_p^2 w_p^2(m)$, $(\Delta^2)^T \mathbf{w}^2 = [0.18, 0.06, -0.04, 0.29, -0.62, -0.16]$, Repositioned $\{(\Delta^2)^T \mathbf{w}^2\} = \begin{bmatrix} 0.18 & 0.06 \\ -0.04 & 0.29 \\ -0.62 & -0.16 \end{bmatrix}$, following $\mathbf{o}_F^1 \rightarrow \mathbf{o}^1$ 	
<ul style="list-style-type: none"> Repositioning the elements of $\Delta_k^1 = [(\Delta^2)^T \mathbf{w}^2]$ at the positions defined by $\mathbf{M}_k^{\text{MP}} \odot \mathbf{M}_k^{\text{ReLU}}$ $\Delta_1^1 = [0 \ 0.18 \ 0 \ 0 \ 0 \ 0.06]^T$, $\Delta_2^1 = [0 \ 0 \ 0 \ 0 \ 0 \ 0.29]^T$, $\Delta_3^1 = [0 \ -0.62 \ 0 \ 0 \ 0 \ -0.16]^T$ where \odot is the Hadamard element-by-element product. 	
<ul style="list-style-type: none"> Gradient in the convolutional layer, $g_k^1(m) = \Delta_k^1(m) * x(m)$, $\mathbf{g}_k^1 = \Delta_k^1 * \mathbf{x}$ $\mathbf{g}_1^1 = [-0.03 \ 0.03 \ 0.06]^T$, $\mathbf{g}_2^1 = [0.01 \ 0.20 \ 0.12]^T$, $\mathbf{g}_3^1 = [0.06 \ -0.02 \ 0.06]^T$ 	
<ul style="list-style-type: none"> Weight update in the convolutional layer $w_k^1(m) \leftarrow w_k^1(m) - 0.1 g_k^1(m)$ $\mathbf{w}_1^1 = [-0.06 \ -0.01 \ -1.47]^T$, $\mathbf{w}_2^1 = [0.45 \ 0.13 \ -0.31]^T$, $\mathbf{w}_3^1 = [1.12 \ -1.01 \ -1.84]^T$ 	
<ul style="list-style-type: none"> Bias update, $b_k \leftarrow b_k - 0.05 \sum_m \Delta_k^1(m)$ $\mathbf{b} \leftarrow \mathbf{b} - 0.05 ([1 \ 1 \ 1 \ 1 \ 1 \ 1] \Delta^1)^T = \mathbf{0} - 0.05 [0.24 \ 0.29 \ -0.78]^T$. 	
<ul style="list-style-type: none"> New iteration with the new signal, $\mathbf{t} = [0, 1]$, $\mathbf{x} = [-0.10 \ -0.12 \ -0.05 \ -0.24 \ 0.89 \ -0.35 \ -0.02 \ -0.00]^T$, Go back to the first step with the new (updated) weights, \mathbf{w}^1 and \mathbf{w}^2, and bias \mathbf{b}. 	

(a) After the first training cycle is finished, as outlined step-by-step in the table above, the process is repeated with a new input noisy signal \mathbf{x} , randomly assuming feature₁ or feature₂, at a random position within the signal, as shown in Fig. 7(a).

The following parameters during the training process are given in Fig. 7:

- The obtained probabilities, P_k , $k = 1, 2$, at the output of the CNN (being the output of the Softmax layer) are given in the second panel of Fig. 7 using black "+" for the values when the correct result should be $P_k = 1$, that is, the value of P_1 is shown when the feature₁ is present and the value P_1 is given by this mark when the feature₂ is present in the input signal. In an ideal case all black "+" should be in positions where this value is equal to 1. The output values P_k are designated by green ".", when the correct output result should be $P_k = 0$, that is, this mark is used for P_1 when the feature₂ is present and for P_2 when the feature₁ is present in the input signal. In an ideal case all green "." should be in positions where this value is equal to 0. The output signals (probabilities) are presented using marks "." and "+" in such a way that the correct positions of the mark "." would always be 0 and the correct position of the mark "+" would always be 1.
- The values of weights in the fully connected layer, during the training process, are given in the third panel in Fig. 7.
- The training process is performed using 200 random realizations of the input signal, randomly assuming feature₁ or feature₂. This cycle of 200 realizations is called an epoch. Then the same set of 200 random realizations is repeated 10 times (then epochs are used in training), that is the CNN was trained over 10 epochs, with no max-pooling used.
- After the CNN is trained in 10 epochs of 200 random realizations of the signal, the update process of the weights in all layers is stopped, and the achieved weights are tested on 100 new random realizations. The results are shown in the last panel in Fig. 7. We can see that the decision was correct in all 100 new cases, where the marks black "+" and green "." are used in the sense described in the first item of this list.

(b) The same setup in Fig. 7 was next used in a CNN with the max-pooling operation, using the factor $P = 3$. The results are shown in Fig. 8 with the same explanation as in Fig. 7. Observe that without max-pooling, the probabilities separate after 600 iterations (3 epochs), while in the case with max-pooling 800 iterations (4 epochs) are needed.

The number of weights when no max-pooling is used, was $((N - M + 1)K) \times 2 = 24 \times 2 = 48$, while with max-pooling it was $((N - M + 1)K/P) \times 2 = 6 \times 2 = 12$. In the case without max-pooling, we used $K = 4$ channels, while in the case of max-pooling, the number of channels was reduced to $K = 3$.

(c) Finally, the same signal was used to train a CNN with one convolutional layer and two fully connected layers, with $K = 5$ channels in the convolution and $P = 3$ being used in max-pooling operation. The number of input neurons in the second fully connected layer was $N_2 = 4$. The Softmax with two output neurons was used for the decision. The results are shown in Fig. 9, with the same notation as in the previous figures. We can see that the convergence of the weights was faster than in the previous two cases. After 300 training cycles

the weights assumed almost steady values. The testing of this network on 100 new random realizations of the input signal was 100% successful, as observed from the last panel in Fig. 9.

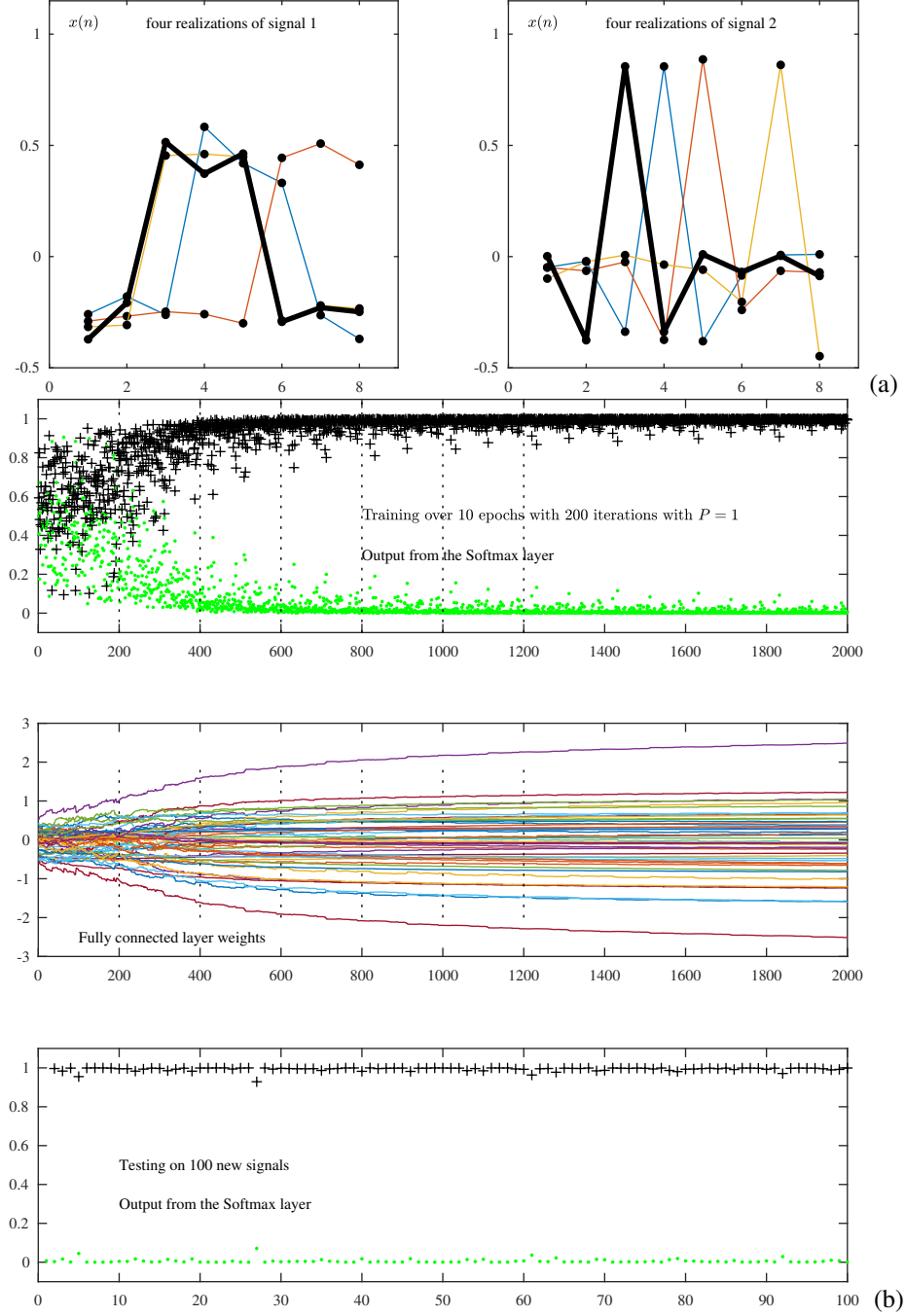


Fig. 7. Operation of a CNN similar to that shown in Fig. 5, with one convolution layer and one FC layer, with two neurons at the output (softmax) layer, max-pooling with $P = 1$ (no max-pooling) and $K = 4$ channels. The FC layer had therefore $((N - M + 1)K) \times 2 = 24 \times 2 = 48$ weights. (a) Several random realizations of the input signal are used for the CNN training and testing. (b) Evaluation of the network output over the training process. (c) Evaluation of the FC layer weights over the training process. (d) Output of the network in the testing stage. Observe the very accurate operation of the trained CNN

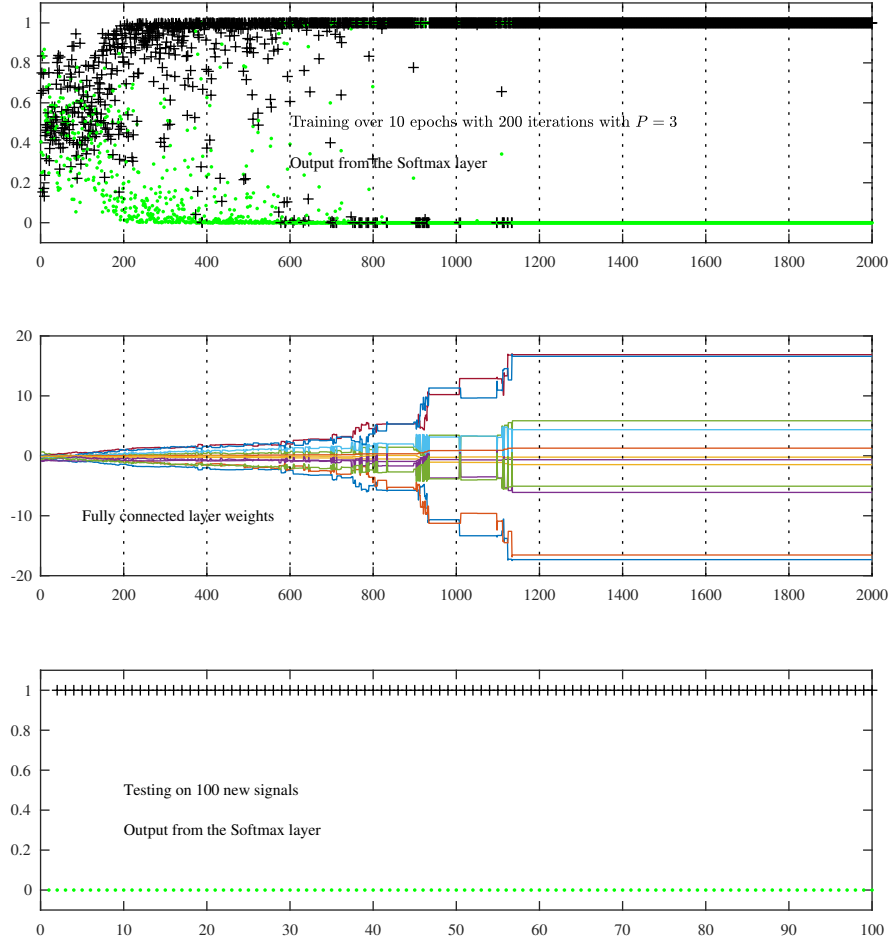


Fig. 8. Illustration of the operation of a CNN with one convolutional layer and one FC layer, with two neurons at the output (Softmax) layer, max-pooling with $P = 3$, and $K = 3$ channels. This helped reduce the number of weights in the FC layer to only $((N - M + 1)K/P) \times 2 = 6 \times 2 = 12$ weights.

V. ADDITIONAL CONSIDERATIONS

Convolutions with 1 in signals or 1×1 filters in images. From Fig. 4 we can see that the number of weights in filters is increased K times for K output signals $o_p^l(n)$, for channels $p = 1, 2, \dots, K$. This new channel dimension increases the number of weights K times. Therefore, the number of parameters increases linearly with the number of convolutions (filter patterns), K , but can be reduced using the so called 1 filters in signals or 1×1 filters in images. We will consider here the simplest and the most commonly used case that reduces this dimension from K to 1. To this end consider Fig. 4, with the filter of length $M = 1$ and a given k , with the weights $w_{k,p}^2(0)$, $p = 1, 2, \dots, K = 4$, for every $k = 1, 2, \dots, K_2$. Then, we obtain just one dimensional output $y_k^2(n)$, $k = 1, 2, \dots, K_2$. Next, K_2 filters be applied, as in Fig. 3, to this one-dimensional signal, to produce the resulting convolutional output. This approach is called *convolutions with 1* and may significantly reduce the number of required weights. Indeed, the number of weights for filters of width M_2 was $M_2 K K_2$, while if convolutions with $M_2 = 1$ are used first, then initially we have $K K_2$ filters 1, to reduce the dimension K to 1, and then $M_2 K_2$ filters for the convolution of signals obtained in such a way. In total, the reduction is significant, since $K K_2 + M_2 K_2 = (M_2 + K) K_2 < (M_2 K) K_2$.

Of course, we may use different lengths of the 1×1 filter in the direction p to reduce, or even increase, the number of weights (if zero-padding is added).

Dropout for Regularizing Deep Neural Networks. Given a large number of neurons and layers, deep neural networks are likely to quickly *overfit* a training dataset. Within the help of the convolutional layers of a CNN, this problem is reduced through max-pooling or output down-sampling (stride). In both cases, the outputs which are ignored in the next layer are either defined by the signal and filters (max-pooling) or by a regular down-sampling scheme (stride). In deep neural networks, regularization is commonly achieved by *randomly dropping out nodes* in the network, whereby every node is considered as a candidate to be dropped out (ignored) with probability P . Then, a deep neural network is trained by means a large number of neural networks, with different architectures, operating in parallel, obtained by different random dropped nodes.

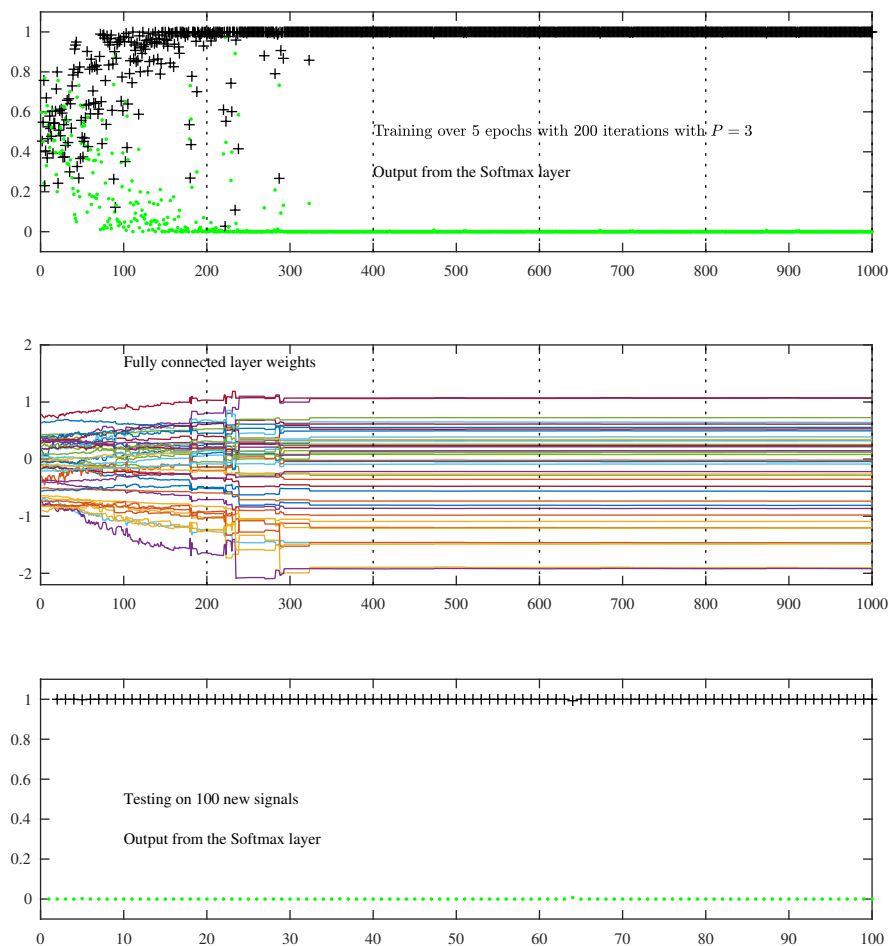


Fig. 9. Illustration of the operation of a CNN from Fig. 5 with one convolution layer and *two* FC layers, with two neurons at the output (softmax) layer, max-pooling with $P = 3$, and $K = 5$ channels. The number of neurons between two FC layers is $N_2 = 4$. This reduces the number of weights in the FC layer to $((N - M + 1)K/P) \times N_2 + N_2 \times 2 = 48$ weights. High training accuracy was achieved with 5 epochs over 1000 iterations. The evolution of weight updates in the first FC layer are shown in the middle panel, while the bottom panel shows the overall network output for test data.

The effect of the neuron drop-out is such that during the training, each update within a layer is performed with a different “view” of the configured layer, forcing nodes within a layer to probabilistically take on more or less responsibility for the inputs and co-adapt to correct mistakes from the previous layers, in turn making the model more robust. In this sense, neuron dropout represents a kind of a sparse activation function from a given layer.

A CNN with dropout can be implemented in the same way as the above described approaches. Notice that since several networks are trained in parallel, a normalization of the weights from each architecture, with probability P , should be performed.

VI. CONCLUSION

We have employed the matched filtering paradigm as a “mathematical lens” to demystify the operation and learning in Convolutional Neural Networks (CNN). A close examination of the convolutional layer within CNNs has revealed a direct and intuitive link with matched filtering for finding the features (patterns) in data. Such a framework has allowed us for a seamless transition between matched filtering and feature identification, together with a unifying and a straightforward platform for understanding the information flow in learning and optimal parameters selection. The material is supported by a comprehensively evaluated example, with detailed numerical outputs and visualizations. This approach has been shown to permit the introduction of CNNs in a theoretically well founded and physically meaningful way, which is beneficial for many communities that do not rely on black box approaches. In addition, the material may be useful in lecture courses in statistical signal processing, machine learning, and statistics, or indeed, as an interesting step-by-step guide to CNNs for the intellectually curious and generally knowledgeable reader.

REFERENCES

- [1] M. H. Hassoun *et al.*, *Fundamentals of artificial neural networks*. MIT press, 1995.
- [2] B. Yegnanarayana, *Artificial neural networks*. PHI Learning Pvt. Ltd., 2009.
- [3] K. Gurney, *An introduction to neural networks*. CRC press, 2018.
- [4] C.-C. J. Kuo, "Understanding convolutional neural networks with a mathematical model," *Journal of Visual Communication and Image Representation*, vol. 41, pp. 406–413, 2016.
- [5] D. Mandic and J. Chambers, *Recurrent neural networks for prediction: Learning algorithms, architectures and stability*. Wiley, 2001.
- [6] S. Kiranyaz, O. Avci, O. Abdeljaber, T. Ince, M. Gabbouj, and D. J. Inman, "1D convolutional neural networks and applications: A survey," *Mechanical Systems and Signal Processing*, vol. 151, p. 107398, 2021.
- [7] C.-C. J. Kuo, "The CNN as a guided multilayer RECOs transform [lecture notes]," *IEEE signal processing magazine*, vol. 34, no. 3, pp. 81–89, 2017.
- [8] A. Ghosh, A. Sufian, F. Sultana, A. Chakrabarti, and D. De, "Fundamental concepts of convolutional neural network," in *Recent Trends and Advances in Artificial Intelligence and Internet of Things*, pp. 519–567, Springer, 2020.
- [9] Y. Li, Z. Hao, and H. Lei, "Survey of convolutional neural network," *Journal of Computer Applications*, vol. 36, no. 9, pp. 2508–2515, 2016.
- [10] Q. Zhang, M. Zhang, T. Chen, Z. Sun, Y. Ma, and B. Yu, "Recent advances in convolutional neural network acceleration," *Neurocomputing*, vol. 323, pp. 37–51, 2019.
- [11] S. Albawi, T. A. Mohammed, and S. Al-Zawi, "Understanding of a convolutional neural network," in *2017 International Conference on Engineering and Technology (ICET)*, pp. 1–6, IEEE, 2017.
- [12] K. O'Shea and R. Nash, "An introduction to convolutional neural networks," *arXiv preprint arXiv:1511.08458*, 2015.
- [13] K. H. Jin, M. T. McCann, E. Froustey, and M. Unser, "Deep convolutional neural network for inverse problems in imaging," *IEEE Transactions on Image Processing*, vol. 26, no. 9, pp. 4509–4522, 2017.
- [14] C. Dong, C. C. Loy, and X. Tang, "Accelerating the super-resolution convolutional neural network," in *European conference on computer vision*, pp. 391–407, Springer, 2016.
- [15] U. R. Acharya, S. L. Oh, Y. Hagiwara, J. H. Tan, M. Adam, A. Gertych, and R. San Tan, "A deep convolutional neural network model to classify heartbeats," *Computers in biology and medicine*, vol. 89, pp. 389–396, 2017.
- [16] P. Kim, "Convolutional neural network," in *MATLAB deep learning*, pp. 121–147, Springer, 2017.
- [17] L. Stankovic and D. Mandic, "Understanding the basis of graph convolutional neural networks via an intuitive matched filtering approach," *arXiv preprint arXiv:2108.10751*, 2021.
- [18] L. Stanković, *Digital Signal Processing with Selected Topics*. CreateSpace Independent Publishing Platform, An Amazon.com Company, 2015.
- [19] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, pp. 1097–1105, 2012.