Published in: IEEE Transactions on Systems, Man, and Cybernetics: Systems (Volume: 53, Issue: 6, June 2023) IEEE DOI: 10.1109/TSMC.2022.3228597

Convolutional Neural Networks Demystified: A Matched Filtering Perspective Based Tutorial

Ljubiša Stanković, *Fellow, IEEE,* Danilo Mandic, *Fellow, IEEE*

***Abstract*—Deep Neural Networks (DNN) and especially Convolutional Neural Networks (CNN) have revolutionized the way we approach the analysis of large quantities of data. However, the largely ad-hoc fashion of their development, albeit one reason for their rapid success, has also brought to light the intrinsic limitations of CNNs - in particular those related to their black box nature. In addition, the ability to 'explain' both the way such systems behave and the results they produce is increasingly becoming an imperative in many practical applications. Therefore, it would be particularly useful to establish physically meaningful mechanisms underpinning the operation of CNNs, thus helping to resolve the issue of interpretability of the processing steps and explain their input-output relationship. To this end, we revisit the operation of CNNs from first principles and show that their very backbone – the convolution operation – represents a matched filter which examines the input for the presence of characteristic patterns in data. Our treatment is based on temporal signals, naturally generated by physical sensors, which admit rigorous analysis through systems science. This serves as a vehicle for a unifying account on the overall functionality of CNNs, whereby both the convolution-activation-pooling chain and learning strategies are shown to admit a compact and elegant interpretation under the umbrella of matched filtering. In addition to helping reveal the physical principles underpinning CNNs and providing an intuitive understanding of their operation, the treatment of CNNs from a matched filtering perspective is also shown to offer a platform to support further developments in this area.**

I. Introduction

Our world is becoming rapidly dependent on data of increasing complexity, diversity and volume; these are generated by readily available sensors, such as signal and image streams from microphones and cameras in multimedia communication and social networks, and increasingly from internet-enabled autonomous electronic devices, e.g. the Internet of Things (IoT). The usefulness of these data streams to us humans is limited by the inevitable bottleneck in both the processing and storage of such exceedingly high data volumes known as the Curse of Dimensionality (CoD) [1], [2], [3].

Learning machines based on Convolutional Neural Networks (CNN) seek to mitigate the issues related to Curse of Dimensionality through the exploitation of local information in data. Consideration of local information is physically justified, as real-world data typically exhibit some sort of smoothness, that is, a degree of similarity among the neighboring samples in signals or pixels in images. Such local information is naturally accounted for in the form of localized patterns in data, known as features; in this way the learning task boils down to performing a search for specific characteristic features in data. Another advantage of operating in the feature domain is that this resolves the inadequacy of standard approaches related to a change of the position of patterns in data, for example, due to translation. Indeed, if a certain feature in e.g. an input image changes its position, then a standard *"brute force"* NN approach will treat such a pattern as a completely different set of pixels, while *"feature based"* learners can be designed to account for certain invariances of the feature space, such as translations, and will look for the specific shapes/patterns of interest anywhere in the analysed data. Much like with standard (fully connected) DNNs, this approach comes at the expense of the inability to understand the underlying mechanism involved, giving rise to an aura of mysticism around big data analysis, as well as making the data processing unnecessarily black box in nature and cumbersome to develop [4], [5], [6].

In the particular case of Convolutional Neural Networks [7], [8], [9], the characteristic patterns (features) of interest are encoded in the so called *convolution filters or convolution kernels* through the training process of CNNs. An example of a simple CNN which consists of a convolutional layer, activation and max-pooling stages, and the fully connected (FC) output layer [10], [11], [12] is shown in Fig. 1. More precisely, the presence of such characteristic patterns is determined by comparing the convolution filters to local patterns in the data, whereby the mechanism for feature/pattern identification is invariant to the change in the position/orientation of features [13], [14], [15], while the feature matching process is performed over the whole signal or image [16], [17], [18], [19]. For more details on a corresponding approach to graph convolutional neural networks, please see our sister paper [20].

To bring the treatment of CNNs closer to the communities working on systems science and at the same time help resolve the interpretability issues arising from their black box nature, we revisit the operation of CNNs by taking inspiration from the theory of matched filters. The convolution of a considered signal with the feature of interest is then used as a mechanism to confirm both the existence and location of a characteristic feature within the analysed data. We further show that such an approach allows for a *unifying perspective of CNNs*, whereby all steps in their operation, such as the *convolution–activation–pooling chain* in the forward pass and the back-propagation based learning strategy, can be considered and fully explained through the lenses of matched filtering. Such a perspective is shown to be physically meaningful, and serves as a basis for a step-by-step visualisation of every stage in CNN operation, based on a matched filtering interpretation of the identification and classification of characteristic patterns in real-world noisy data. It is our hope that the presented approach will help demystify the operation of CNNs, while at the same time helping enhance their interpretability through solid theoretical justification behind the key steps of their operation – thus serving as a basis for their further development.

L. Stanković is with the University of Montenegro, Podgorica, Montenegro. D. Mandic is with Imperial College London, London, United Kingdom. Contact e-mail: ljubisa@ucg.ac.me

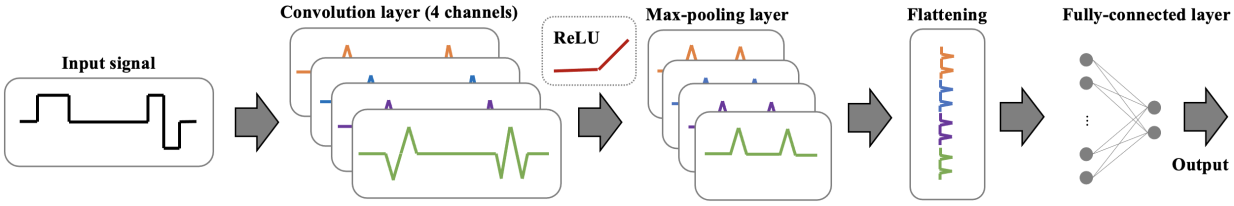

Fig. 1. A simple CNN which consists of an input layer, a convolutional layer with four convolution filters, the nonlinear activation (ReLU) stage, the max-pooling stage, the flattening stage, and the fully connected output layer.

II. Principle of Matched Filtering

Matched filtering is a widely used technique for the detection of a known feature (template), $w(n)$, in an observed noisy signal, $x(n)$, and has found application in areas including radar, sonar, digital communications, and biomedical engineering [21], to mention but a few. A matched filter makes decision based on the cross-correlation between the observed signal, $x(n)$, and the template of interest, $w(n)$, that is based on

$$y(n) = \sum_m w(m)x(n+m), \tag{1}$$

whereby the output of a matched filter, that is, the cross-correlation, $y(n)$, reaches its maximum when a pattern present in the received data stream, $x(n)$, is aligned with the pre–defined feature (pattern) of interest, $w(n)$.

The implementation of the cross-correlation in (1) can be awkward for real-time streaming data and/or large-size features. On the other hand, such a scenario perfectly suits standard digital filters, where the input-output relation represents a convolution between the input, $x(n)$, and the impulse response, $h(n)$, denoted by $y(n) = x(n) * h(n)$ and given by

$$\begin{aligned} y(n) &= \sum_m h(m)x(n-m) = \sum_m x(m)h(n-m) \\ &= x(n) * h(n) = h(n) * x(n). \end{aligned} \tag{2}$$

A comparison between the cross-correlation in (1) and the convolution in (2) immediately suggests a way to implement cross-correlation through convolution. This is achieved by time-reversing the template of interest, $w(m) \to w(-m)$ in (1), which then serves as the "impulse response" in the convolution sum in (2), that is, $h(n) = w(-n)$. Therefore, the convolution-based implementation of the matched filter in (1) has the form

$$\begin{aligned} y(n) &= x(n) * w(-n) = \sum_m w(-m)x(n-m) \\ &\overset{-m\to m}{=} \sum_m w(m)x(n+m) = x(n) *_c w(n). \end{aligned} \tag{3}$$

where the symbol $*_c$ designates the convolution with a time-reversed feature/template vector, $w(-n)$. Figure 2 illustrates the principle of matched filtering in a noise–free scenario, and its implementation through convolution and thresholding.

Remark 1: The translation-invariance property of the convolution operator dictates that, as desired, convolution-based feature detection is independent of the feature position within the considered signal, $x(n)$. This is because $y(n) = x(n) * w(-n)$ is calculated by sliding the window (filter/kernel/pattern), $w(-n)$, of length M along the signal, $x(n)$, and taking a dot-product with the corresponding portion of $x(n)$, at each n. By its very nature, the dot product represents a similarity measure (cosine similarity) which further justifies the use of the convolution operator when searching for patterns in data.

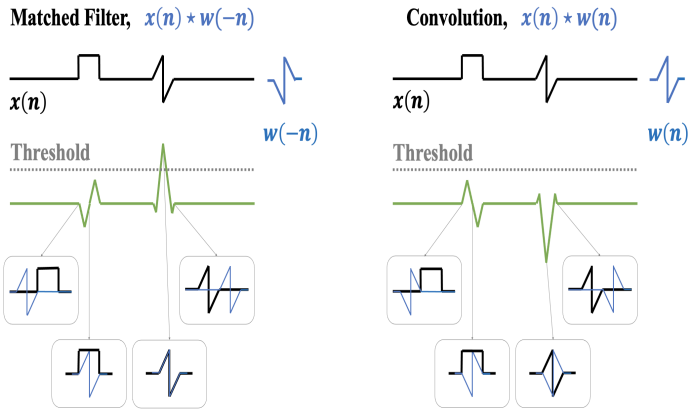

Fig. 2. Operation of a matched filter, which confirms the presence and position of a known triangular template (thin blue line), $w(n)$, in the observed signal, $x(n)$ (thick black line). The matched filter performs cross-correlation between a known template of interest, $w(n)$, and the unknown input, $x(n)$. This cross-correlation is implemented through a convolution between $x(n)$ and a time-reversed template, $w(-n)$, which serves as the impulse response.

Remark 2: We have seen that when using a digital filter to implement the matched filtering operation, the convolution $x(n) * w(-n)$ corresponds to the cross-correlation between the observed signal, $x(n)$, and the feature of interest, $w(n)$, rather than representing their actual convolution, $x(n) * w(n)$. Therefore, the convolutional layer in CNNs (see Fig. 1) performs precisely the matched filtering operation, described in (3) and Figures 2 and 3. Yet, despite this principle (of pattern matching) underpinning the operation of CNNs, *these intrinsically cross-correlational neural networks are referred to as convolutional neural networks*. Indeed, all notations in the literature, such as $\mathbf{x} * rot180^0(\mathbf{w})$ or $\text{conv}\{\mathbf{x}, reverse(\mathbf{w})\}$, assume that the convolution is only applied after the feature vector is time reversed, that is, $x(n) * w(-n)$. We will use a simplified notation $\mathbf{x} *_c \mathbf{w}$, to indicate that the second signal in the convolution is reversed. Another aspect which reinforces our matched filtering perspective of CNNs is that standard convolution often represents a signal conditioning operation (e.g. low-pass filtering) whereby every sample of the output is important; on the contrary, the only output sample of interest from a matched filter is the maximum out of the $N + M - 1$ output samples. This will be used in Remark 4 to interpret the max–pooling operation within CNNs.

III. Finding multiple patterns in data by CNNs

Consider a signal which contains one waveform which belongs to a set of all possible pre-defined templates/features of interests; such as set is also called a dictionary or an alphabet. Fig. 2 shows that the maximum cross-correlation between the received signal and templates from the dictionary will indicate the presence of one of the templates from the dictionary in the signal, together with its location in time. Based on (3), this cross-correlation can be calculated by passing the received waveform through a bank of matched filters, the impulse responses of which are time-reversed versions of the dictionary templates, as elaborated in Example 1 and Fig. 3.

Example 1. *Consider two noisy signals, $x_1(n)$ and $x_2(n)$, shown as black lines in the top two panels in Fig. 3, which contain different features of interest, shown in green. According to Remark 1, the impulse responses of the convolutions, which implement the corresponding matched filters, are the time-reversed versions of the original features (in green) from the top panel, and are designated by the corresponding red and blue lines in the middle panel of Fig. 3. Both noisy input signals are next convolved with these time reversed features according to $y(n) = x(n) * w(-n) = x(n) *_c w(n)$, as in (3). The outputs of the so realised matched filters (red and blue filter) are shown in the bottom panels in Fig. 3. The left two panels at the bottom show the outputs of the red and blue matched filter, $y(n)$, when detecting the presence of the first feature, $w_1(n)$, in the noisy input signal $x_1(n)$, while the two right panels at the bottom show the outputs of the red and blue matched filter, $y(n)$, when identifying the presence of the second feature, $w_2(n)$, in the noisy input signal $x_2(n)$. The maxima at the corresponding matched filters indicate that the first input signal indeed contains the red feature (since the output is above the threshold line), while the second input signal contains the blue feature.*

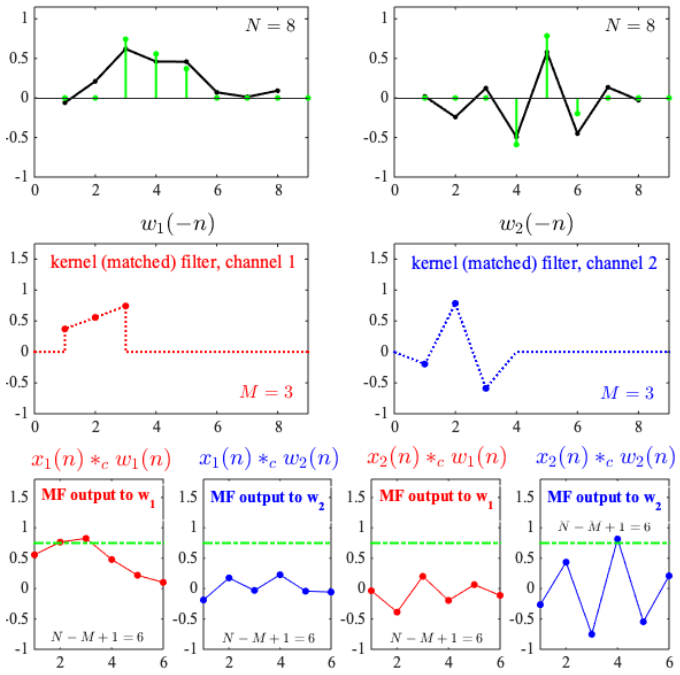

Fig. 3. Illustration of the operation of a matched filter in the presence of noise. *Top panels:* Two noisy input signals, $x_1(n)$ and $x_2(n)$, of length $N = 8$ (with the corresponding noise denoted by $\varepsilon_1(n)$ and $\varepsilon_2(n)$) are shown in solid black line, while the corresponding original noise-free features of interest in the input, denoted by $w_1(n)$ and $w_2(n)$, are of length $N = 3$ and are shown in green line. *Middle panels:* The impulse responses, $w_1(-n)$ and $w_2(-n)$, of the convolutions which implement the corresponding matched filters as in (3), are shown in red and blue. *Bottom panels:* The maxima of the outputs of the red and blue matched filter to the input signals are used to decide the presence of a feature of interest in the input signal, with an appropriate threshold in dashed line. Observe that the feature $w_1(n)$ was correctly detected in the input $x_1(n)$ while the feature $w_2(n)$ was correctly detected in the input $x_2(n)$.

Recall that the impulse response, $h(n) = w(-n)$, in the context of convolution is often referred to as *a filter*; this is the rationale behind the use of the term *convolution filter* when referring to the feature of interest, $w(n)$, within CNNs. Building upon Example 1 and Remark 2, the presence of K distinct features (templates), $w_k(n), k = 1,2,\ldots,K$, in the input signal, $x(n)$, may be identified by employing a bank of K matched filters, the outputs of which are given by

$$y_k(n) = x(n) * w_k(-n) = x(n) *_c w_k(n). \tag{4}$$

The decision on the presence and position of a feature, $w(n)$, in the input signal is then based on the maximum response, $y_{max} = \max_{k,n}\{y_1(n),\ldots,y_K(n)\}$, from this bank of K matched filters. From the definition of matched filter, it then follows that if the input signal, $x(n)$, contains a version of the feature $w_{k_0}(n)$ which is shifted by n_o samples, that is, $w_{k_0}(n - n_0)$, then the maximum of the output of the matched filter, $y_{max} = E_{w_{k_0}} = \sum_n w^2_{k_0}(n) > 0$, will be at time instant $n = n_0$, where $E_{w_{k_0}}$ denotes the energy of the feature $w_{k_0}(n)$. In other words, the bank of convolution filters aims to find

$$k_0 = \arg\{\max_n\{x(n) *_c w_1(n)\},\ldots,\max_n\{x(n) *_c w_K(n)\}\} \tag{5}$$

Remark 3: **(ReLU)** The decision on whether a feature $w_k(n)$ is present in the input $x(n)$ is based on the maximum value of the output of the bank of convolution filters in (4). Such a maximum is equivalent to either the energy of a feature in hand (noise-free case) or the signal to noise ratio (for noisy data) – both positive quantities – so that its location is not affected by a scaling of all kernels, $w_k(n)$, by the same positive factor (even at each iteration). The decision in (4) will remain unaltered if the negative values of every output, $y_k(n)$, $k = 1,\ldots,K$ in (4), are not considered, that is, upon a preprocessing of the form

$$o_k(n) = \text{ReLU}\{y_k(n)\} = \text{ReLU}\{x(n) *_c w_k(n)\} \tag{6}$$

where ReLU stands for *Rectified Linear Unit*, a common nonlinear activation function in CNNs, which is defined by

$$\text{ReLU}(y) = \max\{0, y\}. \tag{7}$$

Remark 4: **(Max–pooling)** Remark 2 shows that when identifying a characteristic feature in the input through matched filtering, as in (5), we are only interested in the value and position of the maximum output sample among all K matched filters. For example, if there was just one feature present in the input signal, that is $K = 1$, then it would be sufficient to retain just one (maximum) sample of the matched filter output for further processing (*cf.* a following layer of a CNN), rather presenting all the outputs of the corresponding convolution sum in (3). The same reasoning applies when searching for

$K > 1$ patterns in data, whereby albeit several "local" samples centered around the maximum of a matched filter do correspond to that same feature, they carry no useful information and can thus be omitted in further processing steps. In the same spirit, our search for features in data will not be compromised if the set of outputs of a matched filter is split into P non-overlapping segments; the next processing step (e.g. the fully connected layer of a CNN in Fig. 1) will consider only such subset-wise maxima when searching for features in data, thus yielding a P-fold reduction in dimensionality. This is precisely the principle behind the so called *max-pooling* operation in CNNs which underpins their computational efficiency over standard NNs.

Remarks 2–4 have shown that the whole *convolution–activation–pooling* chain within CNNs admits a unified physical interpretation from a matched filtering perspective, with each convolution filter tuned to a different distinct feature in the input signal. This equips CNNs with the ability to learn, in an adaptive way, different forms of feature spaces, making them suitable for robust and efficient analysis of signals and images.

IV. The Forward Propagation Pathway in CNNs

We shall now employ the matched filter perspective, introduced above, in order to enable physically meaningful interpretation of the key algorithmic steps in the *forward propagation* path of CNNs. For simplicity, the weights of the convolution filters are assumed to have been already initialised or calculated. The matched filtering perspective of the weight update through back-propagation is addressed in Section III in the Supplement.

For clarity, and without loss of generality, we consider a generic CNN shown in Fig. 1 and Fig. 1 in the Supplement, where the goal is to classify input signals into distinct non-overlapping sets (categories). The steps in the calculation of the output of convolutional neural networks are elaborated below.

1) **Input.** Consider a signal, $\mathbf{x}$, which consists of N samples, and serves as input to a CNN, that is

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

This input is fed into the first layer of CNNs, which is typically the convolutional layer.

2) **Convolutional layer.** This stage of CNNs employs convolution filters of M elements (*cf.* $M \times M$-element filters for images), with the matched filtering interpretation of convolution filters as in (4). The matched filters in the convolutional layer of CNNs are sometimes referred to as *convolutional kernels*, and their lengths are typically $M = 3$ or $M = 5$. Note that K different convolution filters are required if we are looking for K distinct features in the input, $\mathbf{x}$. The elements of the k-th convolution filter within the first convolutional layer are given by

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

for $k = 1, 2, \ldots, K$, with the superscript $(\cdot)^1$ indicating the first convolutional layer. The corresponding outputs are

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

where $*_c$ denotes the convolution of the time-reversed filter (channel), $\mathbf{w}_k^1$, and the signal, $\mathbf{x}$, (i.e. the cross-correlation as in (3)). For more insight, the element-wise form of this convolution, for $M = 3$, is given by

$$\begin{aligned} y_k^1(n) &= w_k^1(0)x(n) + w_k^1(1)x(n+1) + w_k^1(2)x(n+2) \\ &= \sum_{m=0}^{M-1} w_k^1(m)x(n+m). \end{aligned} \tag{8}$$

Observe that the limits of the above convolution sum are reached when $(n+m) = N-1$ for the input $x(n+m)$ and $m = M-1$ for the feature $w(m)$, so that last element of $y_k^1(n)$ is $y_k^1(N-M)$. For example, with $M = 3$ as the length of a convolution filter, the last element in $y_k^1(n)$ becomes $y_k^1(N-3)$. The output dimension of the k-th convolution filter, $\mathbf{y}_k^1$, is therefore $(N-M+1) \times 1$. With K convolution filters considered, the total number of outputs from the first convolutional layer is therefore $K \times (N-M+1)$.

Remark 5: The total number of parameters (weights), $w_k^1(n)$, in the first convolutional layer of CNNs is equal to the product of the length, M, and the number, K, of convolution filters, that is, $M \times K$. Practical applications of CNNs typically employ $M \ll N$, so that the total number of parameters in the first (convolutional) layer of CNNs is much smaller than in a corresponding fully connected layer of neurons (last layer in Fig. 1), whereby each of the N input signal samples is connected through weights to each of the K neurons, a total of $N \times K$ parameters.

3) **Bias.** Like in standard NNs, a constant bias term may be included at the convolutional layer of a CNN, to yield

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

where b_k^1 denotes the bias term at the k-th convolution filter of the first convolutional layer. The vector form of the output of the first convolutional layer then becomes

$$\mathbf{y}_k^1 = \mathbf{x} *_c \mathbf{w}_k^1 + b_k^1.$$

With the inclusion of the bias term, the total number of coefficients at every convolution filter is increased by one.

Remark 6: We have seen in Remark 5 that the number of weights (parameters) in the convolutional layer of a CNN depends only on the size of the convolution filters. With the bias term included, the total number of weights for time-domain signals is therefore $K(M+1)$.

4) **Zero-Padding.** We have seen from Remark 5 that the output of a convolution filter has $N-M+1$ elements, with N as the length of the input and M as the length of the convolution filter – this is in contrast with standard convolution where the output has $N+M-1$ samples. This is because *the output of convolutions filters within CNNs, as in (8), is calculated only over the available input samples,* $[x(0), \ldots, x(N-1)]^T$, *while standard convolution also uses samples outside* $n = 0, \ldots, N-1$. *These are assumed either zero-valued or a periodic extension of the input (circular convolution).* For illustration, consider the case with $N = 8$ and $M = 3$, so that the input signal samples are $\mathbf{x} = [x(0), x(1), \ldots, x(7)]^T$, and the weights

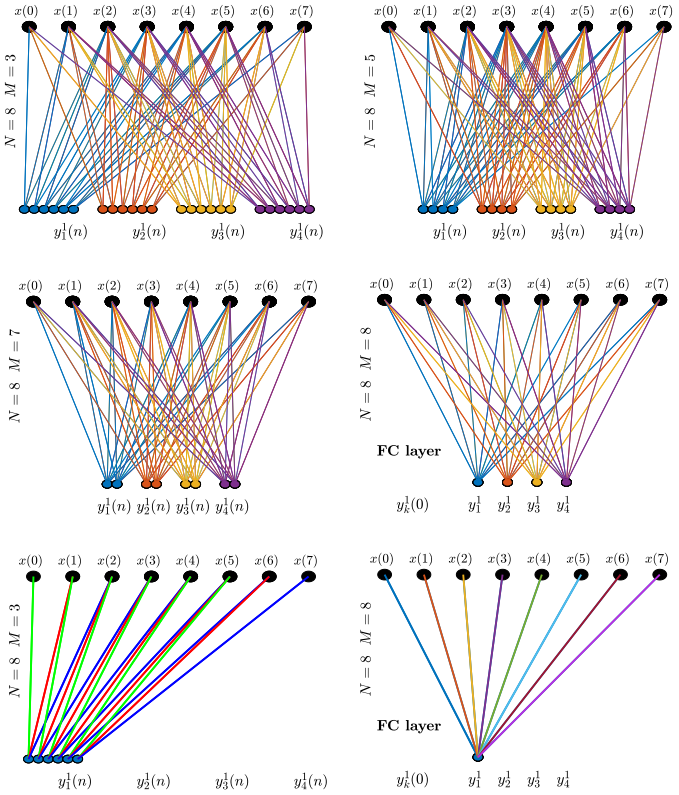

Fig. 4. Operation of the convolutional layer within CNNs, which is illustrated for an input, $x(n)$, of size N=8 samples and with K=4 convolution filters in the convolutional layer (this yields $K=4$ outputs of this layer after the activation and max-pooling operations). Various lengths of the convolution filter, $M \in \{3,5,7,8\}$ were considered. *Top left:* The case with $N=8$ and $M=3$ gives $N-M+1=6$ elements of the output at each convolution filter, $y_1^1(n)$,$y_2^1(n)$,$y_3^1(n)$,$y_4^1(n)$. *Top right:* The case with $N=8$ and $M=5$ gives $N-M+1=4$ elements of the output at each convolution filter $y_k^1(n)$,$k=1,2,3,4$. *Middle left:* The case for $N=8$ and $M=7$ gives two elements at each $y_k^1(n)$. *Middle right:* The case for $M=N=8$ corresponds to the standard FC layer, that is, with $M=N$ the convolutional layer reduces to a fully connected layer, as every convolution filter has now only one element in its output. *Bottom left and Bottom right:* A zoom-in into the two extreme cases with $M=3$ and $M=8$. For clarity, only the weights for the first channel, $w_1(m)$, are shown and are color-coded. Observe the only three different weights $w_1(m)$, $m=0,1,2$ (shown in red, green, and blue) are present for $M=3$, while eight different weights $w_1(m)$, $m=0,1,\ldots,7$ exist for the fully connected layer, with $M=8$.

of the convolution filter $\mathbf{w}_k^1=[w_k(0),\ w_k(1),\ w_k(2)]^T$. Since the convolution filter within CNNs must not use any value of $x(n)$ which lies outside of $n=0,1,\ldots,7$, then according to (8) we establish that:

- The first output sample, $y_k^1(0)$, is obtained by combining $x(0)$, $x(1)$, $x(2)$ with the weights $\mathbf{w}_k^1=[w_k(0),\ w_k(1),\ w_k(2)]^T$.
- The second output sample, $y_k^1(1)$, is obtained by combining $x(1)$, $x(2)$, $x(3)$ with the weights $\mathbf{w}_k^1$.
- The last three "allowed" input signal samples are $x(5)$, $x(6)$, $x(7)$, so that the last output sample is $y_k^1(5)$.
- After calculating $y_k^1(5)$, we have exhausted the available input data, and no more outputs are created.

Fig. 4 shows that since the last available output sample is $y_k^1(5)$, the output of the convolution filter considered has 6 samples, $\mathbf{y}(n)=[y_k^1(0),\ldots,y_k^1(5)]^T$ which is two samples less than the length of the input signal, $N=8$. However, some applications require that the *output of the convolutional layer should be of the same size, N, as the input signal (image)*, instead of having a reduced dimensionality of $N-M+1$. This can be achieved if the input signal (image) is padded with an appropriate number of zeros, which are placed before the first original sample $x(0)$ and after the last original sample $x(N-1)$. In this way, if the input is padded with $M-1$ zeros, then the so augmented input dimension becomes $N+M-1$, and the corresponding dimension of the output of the convolution filter is $(N+M-1)-M+1=N$, which is equal to the input dimension, N. For more detail, see Example 2.

Example 2. ***Zero-padding.*** *Consider a convolutional layer with $M=3$ and $N=8$. To ensure that output of convolutional filters is of the same length as the input, we may add $M-1=2$ zeros which are placed at $x(-1)=0$ and $x(N)=0$. The so "augmented" input signal $\mathbf{x}^a=[x(-1)=0,\ x(0),\ x(1),\ x(2),\ x(3),\ x(4),\ x(5),\ x(6),\ x(7),\ x(8)=0]^T$ yields the convolution output (relative to the original input size)*

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

The first output sample, $y_k^1(0)$, is now obtained by combining $x(-1)=0$, $x(0)$, $x(1)$ with the weights $\mathbf{w}_k^1$ using (8), while the last output sample uses the last three input signal samples $x(6)$, $x(7)$, $x(8)=0$ of the zero-padded input. Notice that after zero-padding with $M-1=2$ zeros, the last output sample becomes $y_k^1(7)$, so that the length of the output of the convolution filter is now the same as the length of the original input signal.
Other strategies for zero-padding include those based on periodic boundary conditions (e.g. for images with cylindrical geometry) and reflection padding [22], to mention but a few.

Remark 7: In general, if the length of the convolution filter is M, the input signal should be padded with $(M-1)$ zeros to yield the "augmented" input length of $N-M+1$ samples and consequently a convolution output of length N. For a convolution filter with an odd number of elements M, the input signal may zero-padded symmetrically, by adding $(M-1)/2$ zero elements before the original starting sample at $n=0$, and $(M-1)/2$ zero elements after the original end sample at $n=N-1$. For this reason the length of the convolution filter, M, in CNNs is typically an odd number, e.g. $M=3,5,7$ as in Fig. 4.

5) **Nonlinear activation function.** Real–world signals are typically nonlinear, while convolution is a linear operation. This calls for a subsequent non-linearity to, for example, restrict the output values to reside within a specified output range, as in the case of sigmoid type of nonlinearities (e.g. logistic or tanh functions) [14]. The most common nonlinear activation function in CNNs is the Rectified Linear Unit (ReLU), defined by (see Remark 3 for its matched filtering interpretation)

$$f(y)=\text{ReLU}(y)=\max\{0,y\} \text{ and } f'(y)=u(y),$$

where $u(y)$ is the unit step function. In the context of CNNs, the ReLU function has several advantages over sigmoidal activation functions, as (i) unlike sigmoid nonlin-

earities, it does not saturate for positive y, and thus yields non-zero gradients for large y, (ii) its calculation is not computationally demanding, even its derivative is a simple to implement unit step function $u(\cdot)$, (iii) in practical applications, networks with the ReLU activation converge faster than those with the saturation-type nonlinearities (logistic, tanh), and (iv) ReLU does not activate neurons with a negative net-input, y, thus performing a kind of *sparsification by deactivation*.

The output of the first convolutional layer, after applying the activation function, now becomes

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

In our example with $M = 3$, the element-wise output of the ReLU becomes

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

The operation of ReLU is elaborated in Example 3.

Example 3. *Consider the output of a convolutional layer within a CNN, given by* $\mathbf{y}_k = \mathbf{x} *_c \mathbf{w}_k^1 + b_k$, *which consists of* $K = 3$ *convolution filters (channels),* $k = 1,2,3$, *for which the element-wise values are given by*

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

The element-wise output from the ReLU activation function then performs a zeroing-out of all negative values in $\mathbf{y}_k$, *to yield*

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

When allocating the correct and accurately aligned proportion of backpropagated errors to the activated neurons in Section III in the Supplement, it is convenient to introduce the **indicator matrix** *which designates the neurons which have been activated/deactivated by ReLU. In our case, the indicator matrix becomes*

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

Since the ReLU produces zero outputs for negative arguments, an issue arises in scenarios with many negative neuron outputs, which makes the indicator matrix, $\mathbf{M}^{ReLU}$, *very sparse thus leaving many neurons without a weight update – the so called* **"dying ReLU"** *phenomenon. This problem can be mitigated using a Leaky ReLU activation, 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 step (down-sampling).** The convolution in (4) is calculated based on shifting the convolution filter one step at a time along the input; this may become prohibitively computationally demanding for large-scale problems. For sufficiently highly sampled (dense) and slow-varying signals, this computational burden may be relaxed by skipping several time instants before calculating the next convolution. This amounts to downsampling the original convolution output, $y(n)$, whereby the degree of downsampling is referred to as the *stride* (step). The stride value of four, $Stride_4$, would thus mean that the convolution is calculated at every fourth time instant (pixel) of the original signal/image, $x(n)$. The stride operator can be applied at various stages of CNN operation, as illustrated in Example 4.

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

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

The reduced-dimension output at a stride of 3 is then 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 the inverse operation of upsampling (inserting zeros) from $\text{Stride}_3\{f(\mathbf{y}_k)\}$ *to the original output* $f(\mathbf{y}_k)$ *is then given by*

$$\mathbf{M}^{Stride_3} = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}^T.$$

Akin to the ReLU indicator matrix in Example 3, the indicator matrix $\mathbf{M}^{Stride_3}$ *serves to correctly align the weight update of CNNs in the backpropagation learning stage (see Supplement).*

7) **Pooling.** In addition to the stride type of downsampling, the output signals at each CNN layer may be further downsampled through the so called *pooling* operation; see also Remark 4. A typical pooling operation employed in CNNs is *max-pooling* which splits the considered signal into non-overlapping P-sample long segments and returns only the maximum value from each such segment (*cf.* returning maximum values from $P \times P$ non-overlapping segments in an image [23]). The output of the *max-pooling* operator over P segments therefore becomes

$$o_k^1(m) = \max\{f(y_k(mP+i)),\ i = 0,1,\ldots,P-1\} \quad (9)$$
$$= F_1\Big(x(n), w(n)\Big),\ m = 0,1,\ldots,(N-M+1)/P.$$

The corresponding indicator matrix, $\mathbf{M}^{MP}$, encodes the positions of the original outputs, $y(n)$, that "survived" max-pooling; its elements are $M_k^{MP}(n) = 1$ for

$$n = \arg\max\{f(y_k(mP+i)),\ i = 0,1,\ldots,P-1\} \quad (10)$$

and $M_k^{MP}(n) = 0$ otherwise, with $n = 0,\ldots,N-M+1$, $k = 1,\ldots,K$. The compact vector form of the output of the max-pooling operator is therefore given by

$$\mathbf{o}_k^1 = F_1(\mathbf{w}_k^1, \mathbf{x}). \quad (11)$$

The so reduced data representation space yields a corresponding reduction in the number of weights in a CNN and the computation burden, as shown in Example 5. Notice that max-pooling also provides approximate translation invariance, as it chooses the maximum value among P neighboring samples, regardless of their position. Another form of pooling is the *average-pooling* operator, whose output represents 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 the original 6 to 2, that is, by the same amount as when employing the stride factor of 3 in Example 4. However, unlike with the stride operation, the positions of the selected samples (and the corresponding upsampling matrix) in max-pooling are signal dependent.

8) **Flattening.** The outputs of the pooling operation, $\mathbf{o}^1$ in (11), are in a vector form. These channel-wise vectors can be concatenated into a *flattened vector*, $\mathbf{o}_F$, of which the elements are (for $k=1,\ldots,K$ and $n=0,\ldots,N-M$)

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

With no max-pooling, this vector is of size $K(N-M+1)$, while after max-pooling with a factor of P, the size of the concatenated (*flattened*) vector, $\mathbf{o}_F^1$, becomes $K(N-M+1)/P$. For images, the two-dimensional max-pooling output is also flattened into a vector [24].
The entirety of the *convolution-activation-pooling-flattening* chain in CNNs is visualised in Example 6.

Example 6. *Consider the input signal with $N=32$ samples, shown in the top panel of Fig. 5, which is fed into the convolutional layer of a CNN consisting of $K=4$ channels (convolution filters) of $M=5$ samples each, followed by the ReLU nonlinear activation function $\max\{0, y_k^1 + b_k^1\}$, $k=1,2,3,4$, and max-pooling with the factor of $P=2$. The initial weights of the convolution filters were Gaussian distributed random numbers (common way of initializing CNNs), while at the max-pooling stage, the signal was grouped into $P=2$ samples long segments, with the largest sample representing the output of this operation. Fig. 5 depicts, in a step-by-step manner, the processing of the input, $x(n)$, by the* convolution-activation-pooling *chain of a simple CNN given in Fig. 1 and Fig. 1 in the Supplement.*

9) **Repeated convolutions.** Some applications may require several repetitions of the convolutional step to search for more complicated, hierarchical features in data; this involves different convolution filter functions (features) at every convolutional layer. Such repeated convolutional may (or may not) employ the activation and pooling functions, while flattening is performed only after the repeated convolutional steps, as illustrated in Example 7.

Example 7. *Consider the CNN from Example 6, but with two successive convolutional layers, whereby the output signals, $o_1^1(n)$, $o_2^1(n)$, $o_3^1(n)$, $o_4^1(n)$ from the first convolutional layer in Fig. 6 are used as inputs to the second convolutional layer. The second convolutional layer comprises $K=5$ convolution 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 with $p=1,2,3,4$. The outputs of the convolution filters in the second layer are 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 yields only the positive outputs, $\max\{0, y_k^2(n) + b_k^2\}$, $k=1,2,3,4,5$, followed by the max-pooling stage with the factor of $P=2$, with $o_1^2(n)$, $o_2^2(n)$, $o_3^2(n)$, $o_4^2(n)$ and $o_5^2(n)$ as the outputs. Finally, the so generated outputs of the second convolutional layer were flattened into a vector, $o_F^2(n)$, which serves as an input to another convolutional layer or the FC layer. These operations, together with the corresponding signal values, are depicted in Fig. 6.*

10) **Fully connected (FC) layers.** The outputs of the convolutional steps, after flattening, serve as inputs to standard fully connected NN layers, as shown in Fig. 1. The FC stage may be a fully connected multilayer structure or a single output layer, as described in Example 8.

Example 8. *An input signal with $N=16$ samples, $x(n)$, is presented to the CNN shown in Fig. 7, which has a (one-dimensional) convolutional layer suitable for temporal data. The signal is processed with $K=4$ convolution filters, $w_1^1(n)$, $w_2^1(n)$, $w_3^1(n)$, and $w_4^1(n)$, each of length $M=5$. The output of these convolution 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, $\max\{0, y_1^1(n) + b_1\}$, $\max\{0, y_2^1(n) + b_2\}$, $\max\{0, y_3^1(n) + b_3\}$, and $\max\{0, y_4^1(n) + b_4\}$. The max-pooling with the factor $P=2$ yields the signals $o_1^1(n)$, $o_2^1(n)$, $o_3^1(n)$, and $o_4^1(n)$, which are combined into the flattened output of the convolutional layer, denoted by $o_F^1(n)$. This signal is used as an input to the first FC layer with 10 neurons, whose outputs feed the second FC layer with two SoftMax output neurons. The overall forward propagation path in a CNN, from the input layer through to the output of the final FC layer, is depicted in Fig. 7.*

V. Initialisation and Back-propagation

All stages of the CNNs training are elaborated and visualized in Example 9. The initialisation and backpropagation procedures are described in detail in Example 2 in the Supplement.

Example 9. *To illustrate training and testing stages of CNNs in a step-by-step manner, consider a two-layer network in Fig. 1 and Fig. 1 in the Supplement, with one convolutional layer and one fully connected layer. The considered input signal had $N=8$ samples, and contained a noisy version of one of the two characteristic patterns (features): (i) a variant of the triangular shape pattern, $feature_1 = [-0.5,\ 1,\ -0.5] + \nu(n)$, for which the target output (class) is $\mathbf{t} = [1,\ 0]^T$ or (ii) a variant of a rectangular three-sample $feature_2 = [1,\ 1,\ 1] + \nu(n)$, for which the target output (class) is $\mathbf{t} = [0,\ 1]^T$, where $\nu(n)$ denotes random uniformly distributed noise whose values lie in the region $[0, 0.3]$. These noisy signals were further corrupted by additive white Gaussian noise with the standard deviation of 0.05, and then normalized to unit energy, as shown in Fig. 8(a).*

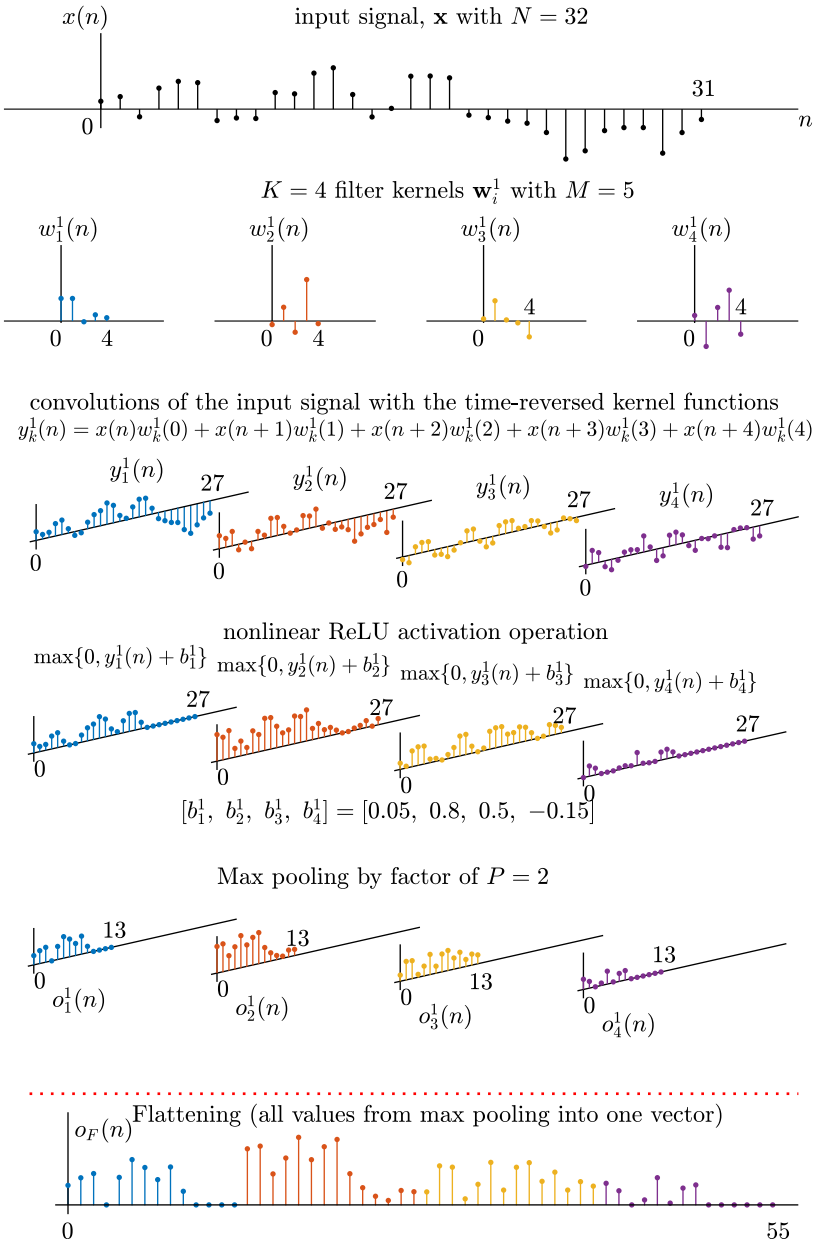

Fig. 5. Visualisation of the convolution-activation-pooling chain in a CNN, with an $N = 32$ sample input, four $M = 5$ sample convolution filters ($K = 4$), the ReLU nonlinear activation, $\max\{0, y_k^1 + b_k^1\}$, $k = 1,2,3,4$, and max-pooling with the factor $P = 2$. The weights of the convolution filter were generated as Gaussian random numbers (common in CNN initialization). The output from the max-pooling stage serves either as input to the next convolutional layer (in CNNs with multiple convolutional layers) or it is flattened to feed neurons of a standard FC neural network layer (our case).

Convolution filters of $M = 3$ samples were used within $K = 3$ channels at the convolutional layer. The SoftMax activation function (see the Supplement) was used at the output of the FC layer, with two output neurons that correspond to the two corresponding patterns in the target signal, **t***, as described above. The network was trained by the backpropagation algorithm, described in Section III in the Supplement. The training stage employed* 200 *random realizations of the input,* **x***, which were presented 10 times to the network, that is, the training was performed over 10 epochs of 200 random signal realizations. The network was tested over* 100 *new random signal realizations, which were not seen by the network during the training stage.*

Details of the forward path of the CNN considered here are given in Fig. 5 - Fig. 7. Step-by-step implementation of the forward and backward calculations are given in the Supplement. Based on the above set-up, the training stage was performed through back-propagation, with the signal values at the different stages of the process given in Example 2 in the Supplement.

(a) After the first training cycle is completed, the process is repeated with a new input noisy input signal, **x***, randomly assuming the presence of either* **feature**$_1$ *or* **feature**$_2$ *at a random position within the signal; some noisy input features are shown in Fig. 8(a). The training and testing stages are visualised in Fig. 8.*

- *The CNN output represents the "probabilities" of the presence of*

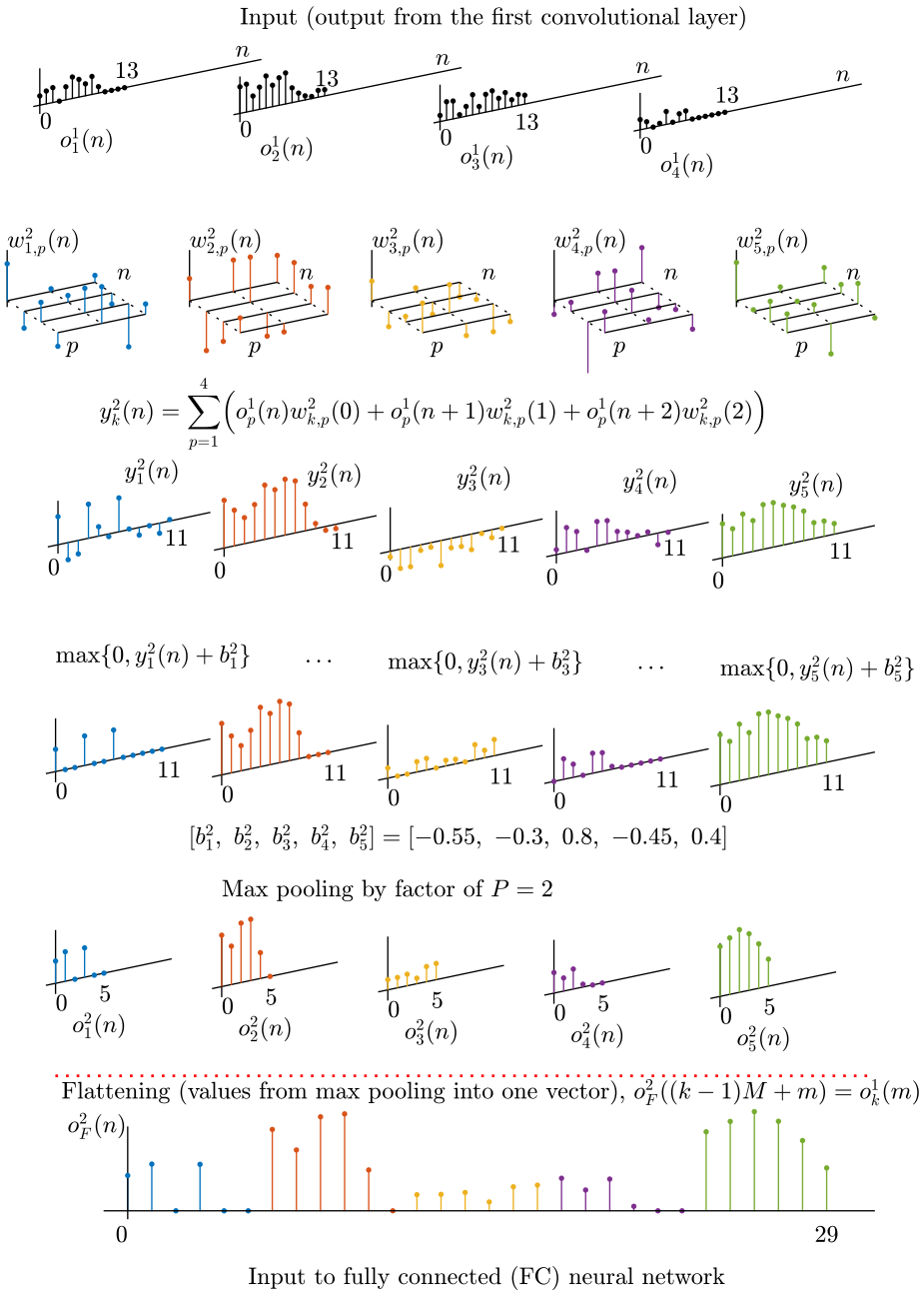

Fig. 6. Operation of a CNN from Example 6, but now with two convolutional layers, whereby the output from the first convolutional layer serves as input to the second convolutional layer. Five convolution filters ($K = 5$) were used in the second convolutional layer, each $M = 3$ samples long, followed by the ReLU nonlinear activation function, $\max\{0, y_i^2 + b_i^2\}$, and max-pooling with the factor of $P = 2$, whereby the signal from the previous step is grouped into segments of two samples with the largest sample serving as the overall output. The signal from the max-pooling stage then serves either as input to the next convolutional layer (in CNNs with multiple convolutional layers) or it is flattened to feed neurons of a standard fully connected (FC) neural network layer (as in our case). The weights of the convolution filter were generated as random Gaussian numbers (a common way of CNN initialization).

the two features in the noisy input, denoted respectively by P_1 and P_2. Therefore, when **feature**$_1$ *is present in the noisy input,* $\mathbf{x}$*, the target signal is* $\mathbf{t}_1 = [1,\ 0]^T$ *and the output of the SoftMax layer in an ideal case should be close to* $P_1 = 1$ *and* $P_2 = 0$*. Regarding the presence of* **feature**$_2$ *in* $\mathbf{x}$*, the corresponding target signal is* $\mathbf{t}_2 = [0,\ 1]^T$ *and the SoftMax outputs should ideally approach* $P_1 = 0$ *and* $P_2 = 1$*. The evolution of the SoftMax output during training is illustrated in Fig. 8(b), where for a consistent account of the overall accuracy:*

- *For the target* $\mathbf{t}_1$*, the black "+" denotes the values of* P_1 *and the green"·" the values of* P_2*,*
- *For the target* $\mathbf{t}_2$*, the black "+" denotes the values of* P_2 *and the green"·" the values of* P_1*.*

In this way, a perfectly trained CNN yields all black "+" marks at 1 and all green "." marks at 0. Note that for the considered scenario, $P_1 + P_2 = 1$ *always holds for the SoftMax outputs.*

- *The evolution of weights in the fully connected layer during the training iterations is depicted Fig. 8 (c). Observe the gradual convergence of all the weights along the training iterations.*
- *The training process was performed as above; the CNN was*

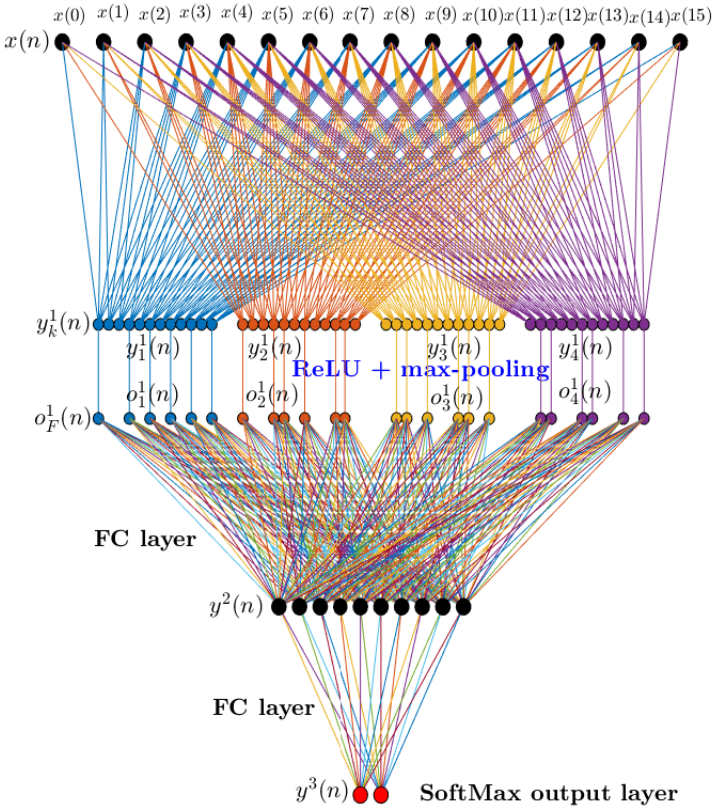

Fig. 7. A simple CNN architecture which is suitable for finding features in temporal signals. It consists of one convolutional layer, a ReLU activation stage, a max-pooling stage, and two FC layers. The second (SoftMax FC layer) has two output neurons. Variants of this architecture are used in Example **??**.

trained over 10 epochs with 200 *random realisations of the noisy input, which contained either* $feature_1$ *or* $feature_2$*, in each epoch, and with no max-pooling employed.*

- *After the training was completed, the performance of the CNN was tested over 100 new random realizations of the noisy input signal, with the results shown in Fig. 8(d). Observe the success of training, as indicated by the correct decisions in all 100 new cases presented to the network, where the black "+" and green "." marks notation is used as described above.*

(b) The same setup as in Fig. 8 was next used within a CNN with the max-pooling operation at the convolutional layer, whereby max-pooling with a factor $P = 3$ *was performed. In the case without max-pooling, we used* $K = 4$ *channels in the convolutional layer, while in the case with max-pooling, the number of channels was reduced to* $K = 3$*. In this way, the number of weights* **with no max-pooling** *was* $((N - M + 1)K) \times 2 = 24 \times 2 = 48$*, compared with* $((N - M + 1)K/P) \times 2 = 6 \times 2 = 12$ **when max-pooling was used**. *The results are shown in Fig. 9.*

(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 convolutional layer and the factor* $P = 3$ *in the 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 again to produce the decision. The results are shown in Fig. 10, using the same notation as above. Observe that in this case the convergence of the weights in the output layer was faster than in Case a) and Case b), as after about 300 training cycles the weights effectively approached their steady values and remained unaltered until the end of training.*

The testing of this network over 100 new random realizations of the input signal was 100% successful.
Full details of every step in the forward and error backpropagation paths are given in Example **??** *in the Supplement.*

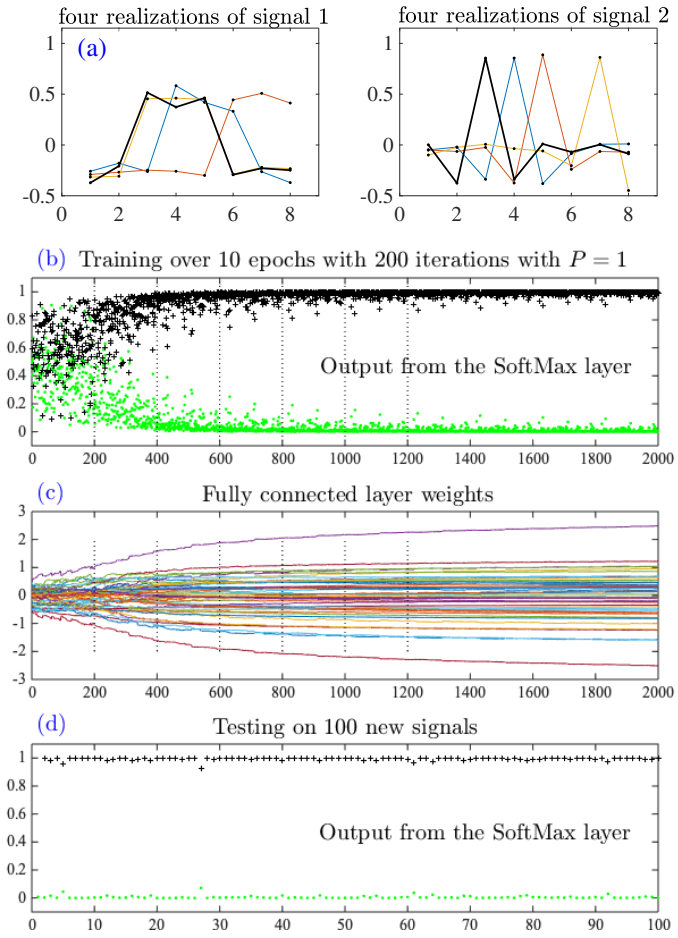

Fig. 8. Operation of a CNN similar to that shown in Fig. 7, which consists of one convolutional layer and one FC layer, with two neurons at the output (SoftMax) layer, max-pooling with the factor $P = 1$ (no max-pooling), and $K = 4$ channels at the convolutional layer. The FC layer had therefore $((N - M + 1)K) \times 2 = 24 \times 2 = 48$ weights. The task was to identify the two features, a rectangular $\mathbf{feature}_1$ and a triangular $\mathbf{feature}_2$ from noisy inputs, $x(n)$. (a) Some random realizations of the inputs used in the CNN training and testing. (b) The output probabilities of the CNN for the task of identification of the presence of a feature (either $\mathbf{feature}_1$ or $\mathbf{feature}_2$) are denoted by a black "+" if the corresponding correct SoftMax output should be equal to 1, and by a green "·" if the corresponding correct SoftMax output should be 0. (c) Evolution of the 48 weights in the FC layer along the training process. Some of the weights (e.g. the top and bottom curve) did not fully converge to their optimal values over the 2,000 training iterations, causing a minor uncertainty at the output. (d) Test results over 100 random realisations of $\mathbf{x}$, with"+" and "·" as above. Observe an almost perfect identification of the two patterns in $\mathbf{x}$.

VI. Dimensionality Reduction Through 1-filters

Repeated convolutions, that is, a sequence of subsequent convolutional layers, are often used in CNNs to detect so-called hiearchical features, however, this comes at the expense of an increase in the dimensionality of convolution filters in

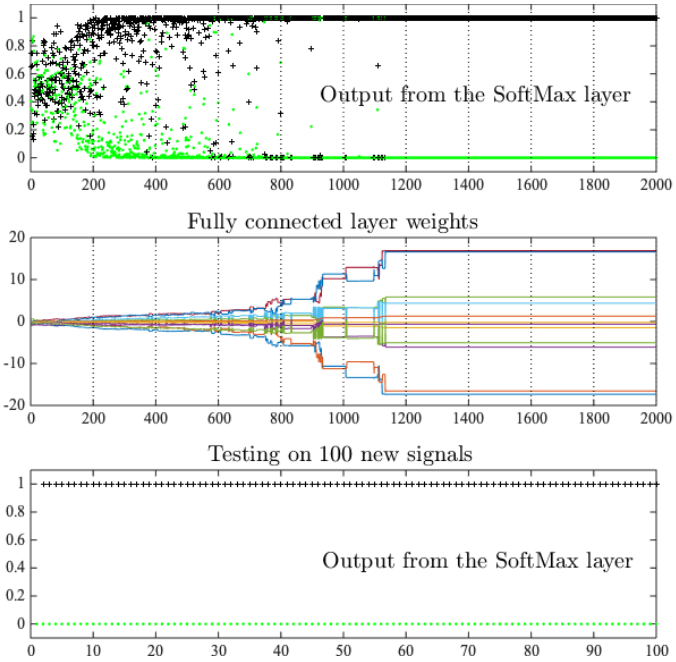

Fig. 9. Illustration of the operation of a CNN with one convolutional layer and one FC layer with two SoftMax output neurons, max-pooling with the factor $P = 3$, and $K = 3$ convolutional channels. This helped reduce the number of weights in the FC layer to only $((N-M+1)K/P) \times 2 = 6 \times 2 = 12$ weights. *Top:* Output probabilities of the CNN for the task of identification of the presence of a feature (either $\mathbf{feature}_1$ or $\mathbf{feature}_2$) in Figure 8 (a) are denoted by a black "+" if the corresponding correct SoftMax output should be equal to 1, and by a green "·" if the correct SoftMax output should be 0. The output of the CNN yields almost perfect probabilities of the detection of the two input noisy patterns after about 1,150 iterations of the training procedure. *Middle:* Evolution of weights of the FC layer shows that they settled to their final values after about 1,150 iterations. *Bottom:* The network output for test data; observe the 100% success in the classification of the two patterns of interest from noisy inputs, which was achieved with 12 weights, only a fraction of the 48 weights required for the case with no max-pooling in Figure 8

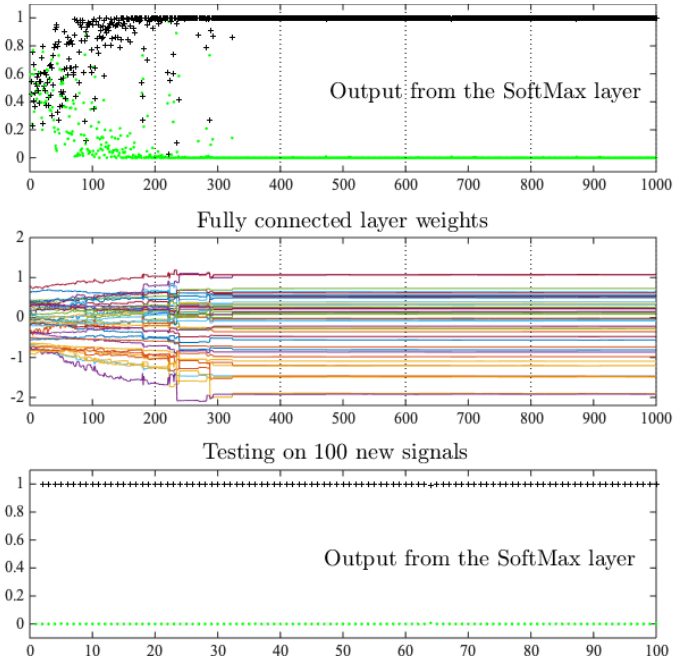

Fig. 10. Illustration of the operation of the CNN from Fig. 7 which consists of one convolutional layer and *two* FC layers, with two neurons at the output SoftMax layer, max-pooling with the factor $P = 3$, and $K = 5$ convolution channels. The number of neurons in each FC layer was $N_2 = 4$. This reduces the number of weights in the FC layer to $((N-M+1)K/P) \times N_2 + N_2 \times 2 = 48$ weights. The outputs were designated as in Figures 8 and 9. *Top:* Perfect training accuracy was achieved over only 5 epochs with 200 iterations of random inputs for each epoch, with the output probabilities converging to their correct values after about 320 training iterations. *Middle:* Evolution of the 48 weights in the first FC layer along the training procedure. The weights converged to their final values after about 320 iterations. *Bottom:* Network output for test data; observe the 100% success in the classification of the two patterns of interest from noisy inputs.

the 2^{nd} and subsequent convolutional layers, compared to the 1^{st} convolutional layer. The increase in the dimensionality of the parameter space with the number of inputs channels to a convolutional layer can also be observed by comparing Fig. 5, which illustrates the operation of the 1^{st} convolutional layer that has a single-channel input, $x(n)$, and Fig. 6 which illustrates the operation of the 2^{nd} convolutional layer that has a multi-channel input, $o_p^1(n), p = 1,2,3,4$.

More precisely, the input, $x(n)$, to the 1^{st} convolutional layer (top panel in Fig. 5) is a vector with N samples, while the input to the 2^{nd} convolutional layer (top panel in Fig. 6) is the output of the 1^{st} convolutional layer (third panel in Fig. 5) which is in the form of $K = 4$ channel-wise output vectors, $o_p^1(n)$, of $N - M + 1$ samples each, that is, a total of $K \times (N - M + 1)$ samples. *In other words, the convolutional kernels within the* 1^{st} *convolutional layer operate in the temporal domain, while the corresponding kernels in the* 2^{nd} *convolutional layer operate in the joint channel-time domain.* This increase in dimensionality quickly becomes computationally prohibitive for multiple successive convolutional layers (repeated convolutions).

A natural way to reduce the parameter space would therefore be to reduce the dimensionality of the input to the repeated convolutional layers. To this end, we may replace the 2^{nd} convolutional layer with a suitably chosen Multiple Input Single Output (MISO) system, to produce a single-channel output from a multi-channel input by means of the so called 1-filter. This is a convolution kernel of width $M = 1$ which effectively takes the weighted average of all input channels (along the channel dimension), for a fixed temporal index, n. In this sense, 1-filters are a special type of dimension-wise convolution, since they operate along the channel dimension only. Notice that dimension-wise convolutions can also be performed across the other dimensions of the input.

Motivated by the principle of separability of multi-dimensional operators into a sequence of dimension-wise operators [2], [25], the concept of *separable convolution* comprises successive dimension-wise convolutions along distinct dimensions of the input in order to reduce the dimensionality of the parameter space. This allows us to separate each 2D channel-time convolution filter in the 2^{nd} layer – designated by the respective indices p and n of the weights $w_{k,p}^2(n)$ – into two independent lower-dimensional (1-dimensional) convolution kernels, of which one operates along the channel dimension, p, and the other along the temporal dimension, n. This is akin to replacing a 2D convolution in images by a sequence of one 1D convolution in the vertical direction, followed by another

1D convolution in the horizontal direction; in this way, a 2D convolution kernel with e.g. $6 \times 6 = 36$ elements would be replaced by two 1D kernels of respective sizes 6×1 and 1×6, a total of 12 elements and a 3-fold reduction in size.

In the same spirit, the operation of the 2^{nd} convolutional layer for signals can be simplified as follows.

1) **Filtering along the p-index (channel dimension).** For every sample index, n, of the input to the 2^{nd} convolutional layer, $o_p^1(n)$, a *weighted average* is performed along the channel index p, $p = 1,2,\ldots,K$. This operation requires only one kernel of K elements per sample index, n. Since there are K_2 convolution filters in the 2^{nd} convolutional layer, we need K_2 different weighted averages (kernels) of this type in order to produce the K_2 inputs into the second stage. The weights of the weighted averaging filters in this step are denoted by $w_{k,p}^{21}(0)$, with $p = 1,2,\ldots,K$ and $k = 1,2,\ldots,K_2$, so that the output of this stage becomes

$$o_k^{21}(n) = \sum_{p=1}^{K} o_p^1(n) w_{k,p}^{21}(0) \qquad k = 1,\ldots,K_2. \tag{12}$$

   Since the filtering is performed along the channel index, p, and not along the time index, n, these weights are channel-dependent but not time-dependent. In other words, the averaging kernels, $w_{k,p}^{21}(0)$, have the effective width of $M = 1$, and are called 1-filters.

2) **Filtering along the n-index (time dimension).** Now that we have "marginalized" the original joint channel-time convolution operation along the channel dimension, we can employ a "standard", temporal, convolutional layer as the second step in the approximation of the 2^{nd} convolutional layer, that is, based on the kernel weights $w_l^{22}(m)$, $l = 1,2,\ldots,K_2$, $m = 0,1,\ldots,M_2 - 1$. In this way, the convolutions are performed along the time index, n, but not along input channel index, p, for each of the K_2 convolution channels. This is because the inputs to this stage have already been averaged out in Step 1. Therefore, the overall output of this stage becomes

$$y_{k,l}^2(n) = \sum_{m=0}^{M_2-1} o_k^{21}(n+m) w_l^{22}(m), \quad k = 1,\ldots,K_2, \;\; l = 1,\ldots,K_2. \tag{13}$$

The above two-step procedure has allowed us to employ two sets of double-indexed filters, $w_{k,p}^{21}(0)$ and $w_k^{22}(m)$, instead of the original single set of tripple-indexed filters, $w_{k,p}^2(n)$, for every convolution channel of the 2^{nd} convolutional layer. Notice that, since in Step 1 the channel dimension has been marginalized out through 1-filters, the temporal convolution channels in Step 2 are shared among all K_2 output signals from the Step 1, as illustrated in Fig. 2 in the Supplement.

Example 10. *To indicate the extent to which convolutions with 1-filters can reduce the number of required CNN weights, recall that the original number of weights in the K_2 convolution filters of width M_2 within the second convolutional later was $M_2 K \times K_2$. If the convolutions with 1-filters are employed after the first convolutional layer, this gives $K \times K_2$ weights, $w_{k,p}^{21}(0)$, of the 1-filters in the first step. The next step then employs K_2 standard convolution filters with weights, $w_k^{22}(m)$, which are of full length M_2, with a total number of the weights $M_2 \times K_2$. This yields a significant reduction in the total number of weights compared to the original $M_2 K K_2$ tripple-indexed weights, since $K K_2 + M_2 K_2 = (M_2 + K) \times K_2 < (M_2 K) K_2$.*

To further depict this property, consider Fig. 6, with $M = 4$, $K = 4$, $M_2 = 3$, and $K_2 = 5$, so that the total number of filter weights in the second convolutional layer is $3 \times 4 \times 5 = 60$. With the use of 1-filters, the number of weights in the second convolutional layer reduces to $4 \times 5 + 3 \times 5 = 35$, as illustrated in Fig. 11.

Remark 8: The operation of convolutions with 1-filters employs the frequently used principle of separability of functions which operate on multiple variables, akin to the separability of a joint pdf of multiple Gaussian random variables into the product of individual distributions or Kronecker separability of multi-indexed tensors [2], [25]. In the same spirit, by means of the above two-step dimension-wise separable convolutions, we have been able to replace the original single set of triple-indexed filters, $w_{k,p}^2(n)$, in the repeated 2^{nd} convolutional layer with two sets of double-indexed filters, $w_{k,p}^{21}(0)$ and $w_k^{22}(m)$. In this way, a joint channel-time filtering at the 2^{nd} convolutional layer has been separated into two dimension-wise operations: (i) a weighted average over the channel index (akin to weighted ensemble average over convolution kernels) and (ii) filtering along the temporal dimension, to yield a considerable reduction in the number of parameters. Of course, convolutions with 1-filters are an approximation, in the same way a 2D filter in images is not equivalent to applying two independent 1D filters separately along the vertical and horizontal axis.

Remark 9: Upon employing the principle of separable convolutions, the output of the 2^{nd} (and other repeated) convolutional layer(s) is produced through the standard temporal convolutions of reduced dimensionality, that is, based on single-channel outputs of the 1-filters and the feature kernels. **This is precisely the matched filtering operation** whereby the convolutional kernels in Step 2 above are matched to the higher-order features in the weighted averaged $o_k^{21}(n)$, as described in (3) and (8), and Example 3. Note that different lengths of 1×1 channel-wise filters may be used in images.

VII. Conclusion

We have employed the matched filtering perspective as a mathematical lens to help demystify the principles of information flow and learning in Convolutional Neural Networks (CNN). A close examination of the convolutional layer within CNNs has revealed a direct and intuitive link with the task of finding features (patterns) in data through matched filters – a common paradigm in systems engineering. Such a perspective has enabled a seamless transition between the well understood and theoretically supported matched filtering paradigm and feature identification mechanisms in CNNs, together with providing a unifying platform for the analysis and interpretation of the various steps in both their forward pass and learning procedures. The matched filtering perspective has been shown, both through analysis and detailed visualization of every step

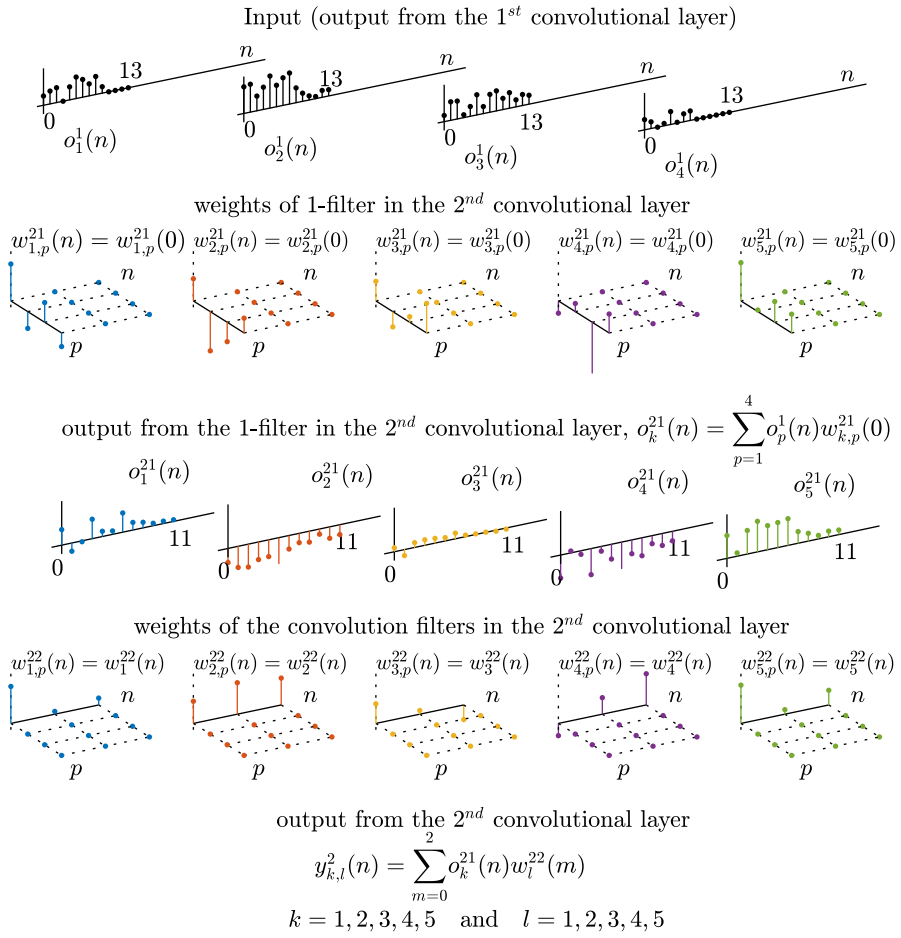

Fig. 11. Illustration of the operation of convolutions with 1-filters in the 2^{nd} convolutional layer from Fig. 6. **Step 1:** The outputs from $K = 4$ channels of the first convolutional layer, $o_p^1(n)$, $p = 1,2,3,4$, undergo a spatial weighted average (across the channel index p and for every fixed time index n) using four-sample (since $K = 4$) weights $w_{k,p}^{21}(0)$, $k = 1,2,3,4,5$. Since there are $K_2 = 5$ assumed channels in the 2^{nd} convolutional layer, this weighted average is performed $K_2 = 5$ times, based on five independent averaging filters to marginalise. The total number of the weights in this step is $KK_2 = 20$. **Step 2:** The so obtained $K_2 = 5$ outputs from Step 1 are used as inputs to $K_2 = 5$ channels of a "standard" convolution filter, $w_k^{22}(m)$, of length $M_2 = 3$, which performs convolution along the time index, n. The total number of weights in this step is $K_2 M_2 = 15$. The total number of adaptive weights in Step 1 and Step 2 is therefore $KK_2 + K_2 M_2 = 35$. On the other hand, if the convolution filters in the 2^{nd} convolutional layer were applied directly to the $K = 4$ outputs of the 1^{st} convolutional layer, as in Fig. 6, we would have had $K_2 = 5$ filters with $K = 4$ outputs each (inputs to the second layer) and the length of every filter would have been $M_2 = 3$, a total $KK_2 M_2 = 60$ adaptive weights as in the case shown in Fig. 6 (second panel).

of their operation, to permit the introduction of CNNs in a theoretically well founded and physically meaningful way. This is likely to be beneficial for research communities that do not rely on black box approaches, such as those working on various aspects of cybernetics, systems science and sensor signal processing. Moreover, the generic nature of the matched filtering perspective on CNNs opens new vistas for further developments in the area and admits generalisation to higher dimensional [26] and irregular domains [27], [25]. In addition to providing a basis for extensions to image-CNNs [23], the material may be useful in lecture courses on CNNs or indeed, as a step-by-step guide for the intellectually curious reader.

REFERENCES

[1] R. Bellman, *Dynamic programming*. Princeton University Press, 1957.

[2] A. Cichocki, N. Lee, I. Oseledets, A.-H. Phan, Q. Zhao, and D. P. Mandic, "Tensor networks for dimensionality reduction and large-scale optimization: Part 1 low-rank tensor decompositions," *Foundations and Trends® in Machine Learning*, vol. 9, no. 4-5, pp. 249–429, 2016.

[3] S. Zhang and A. Constantinides, "Lagrange programming neural networks," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 39, no. 7, pp. 441–452, 1992.

[4] M. Gazda, M. Hireš, and P. Drotár, "Multiple-fine-tuned convolutional neural networks for Parkinson's disease diagnosis from offline handwriting," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 52, no. 1, pp. 78–89, 2021.

[5] S. M. J. Jalali, S. Ahmadian, A. Kavousi-Fard, A. Khosravi, and S. Nahavandi, "Automated deep CNN-LSTM architecture design for solar irradiance forecasting," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 52, no. 1, pp. 54–65, 2021.

[6] T. Li, Z. Zhao, C. Sun, L. Cheng, X. Chen, R. Yan, and R. X. Gao, "Waveletkernelnet: An interpretable deep neural network for industrial intelligent diagnosis," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 52, no. 4, pp. 2302–2312, 2021.

[7] C. Dong, C. C. Loy, and X. Tang, "Accelerating the super-resolution convolutional neural network," in *Proc. European Conference on Computer Vision*, pp. 391–407, Springer, 2016.

[8] 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," *Comp. in Biology and Medicine*, vol. 89, pp. 389–396, 2017.

[9] P. Kim, "Convolutional neural network," in *MATLAB deep learning*, pp. 121–147, Springer, 2017.

[10] S. Albawi, T. A. Mohammed, and S. Al-Zawi, "Understanding of a

convolutional neural network," in *Proc. of IEEE International Conference on Engineering and Technology (ICET)*, pp. 1–6, 2017.
[11] K. O'Shea and R. Nash, "An introduction to convolutional neural networks," *arXiv preprint arXiv:1511.08458*, 2015.
[12] 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.
[13] 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.
[14] D. Mandic and J. Chambers, *Recurrent neural networks for prediction: Learning algorithms, architectures and stability*. Wiley, 2001.
[15] 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.
[16] C.-C. J. Kuo, "The CNN as a guided multilayer RECOS transform [lecture notes]," *IEEE SP Magazine*, vol. 34, no. 3, pp. 81–89, 2017.
[17] 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 IoT*, pp. 519–567, Springer, 2020.
[18] Y. Li, Z. Hao, and H. Lei, "Survey of convolutional neural networks," *Journal of Computer Applications*, vol. 36, no. 9, pp. 2508–2515, 2016.
[19] 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.
[20] L. Stankovic and D. Mandic, "Understanding the basis of graph convolutional neural networks via an intuitive matched filtering approach," *IEEE SP Magazine, in print, arXiv preprint arXiv:2108.10751*, 2022.
[21] L. Stanković, *Digital Signal Processing with Selected Topics*. CreateSpace Independent Publishing Platform, An Amazon.com Company, 2015.
[22] K. M. Iftekharuddin and A. A. Awaal, *Field guide to image processing*. SPIE Press, 2012.
[23] S. Li, X. Zhao, L. Stankovic, and D. Mandic, "Demystifying CNNs for images by matched filters," *arXiv preprint arXiv:4547556*, 2022.
[24] K. Fukushima, "Artificial vision by deep CNN neocognitron," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 51, no. 1, pp. 76–90, 2021.
[25] L. Stanković, D. Mandic, M. Daković, M. Brajović, B. Scalzo, S. Li, and A. G. Constantinides, "Data analytics on graphs Part III: Machine learning on graphs, from graph topology to applications," *Foundations and Trends® in Machine Learning*, vol. 13, no. 4, pp. 332–530, 2020.
[26] C. Tian, Y. Xu, W. Zuo, C.-W. Lin, and D. Zhang, "Asymmetric CNN for image superresolution," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 2021.
[27] Z. Song, X. Yang, Z. Xu, and I. King, "Graph-based semi-supervised learning: A comprehensive review," *preprint arXiv:2102.13303*, 2021.

Ljubiša Stanković (*M'91–SM'96–F'12*) is a professor with the University of Montenegro (UoM) and specializes in Signal Processing. He received his BEng degree in Electrical Engineering from UoM, MSc in communications from the University of Belgrade, and PhD from UoM in 1988. He spent 1984/85 academic year at the Worcester Polytechnic Institute, Worcester, MA, as a Fullbright Fellow. From 1997 to 1999, he was an Alexander von Humbolt Fellow at the Ruhr University Bochum, Germany. During the period of 2003 to 2008, he was the Rector of the University of Montenegro. In 1997 Prof Stanković received the highest state award of Montenegro for his scientific achievements. Prof Stankovic was the Ambassador of Montenegro to the United Kingdom, Ireland, and Iceland from 2011 to 2015, and a visiting academic at Imperial College London in 2012-2013. He has published about 450 technical papers, almost 200 of which are in the leading international journals. He was an Associate and Senior Area Editor of the IEEE Transactions on Image Processing, an Associate Editor of the IEEE Signal Processing Letters, and an Associate Editor of the IEEE Transactions on Signal Processing. Prof. Stankovic is currently Deputy Editor-in-Chief of IET Signal Processing, a member of the Editorial Board of Signal Processing, and a member of the IEEE Signal Processing Society Technical Committee on Theory and Methods. He has been a member of the National Academy of Science and Arts of Montenegro (CANU) since 1996, the Vice-President of CANU since 2016, and a member of the Academia Europaea and the European Academy of Sciences and Arts. Prof Stankovic (with coauthors) won the Best Paper Award from the European Association for Signal Processing (EURASIP) in 2017, the 2020 Best Column Award in IEEE Signal Processing Magazine, and the Outstanding Paper Award at the IEEE ICASSP 2021. Prof Stanković was elected to a Fellowship of the IEEE for his contributions to time-frequency signal analysis.

Danilo Mandic (*M'99–SM'03–F'13*) is a professor with Imperial College London, UK, and has been working in the areas of machine intelligence, statistical signal processing, big data, data anaytics on graphs, bioengineering, and financial modelling. He is a Fellow of the IEEE and a current President of the International Neural Networks Society (INNS). Dr Mandic is a Director of the Financial Machine Intelligence Lab at Imperial, and has more than 500 publications in international journals and conferences. He has published two research monographs on neural networks, entitled "Recurrent Neural Networks for Prediction", Wiley 2001, and "Complex Valued Nonlinear Adaptive Filters: Noncircularity, Widely Linear and Neural models", Wiley 2009, and has co-edited books on Data Fusion (Springer 2008) and Neuro- and Bio-Informatics (Springer 2012). He has also co-authored a two-volume research monograph on tensor networks for Big Data, entitled "Tensor Networks for Dimensionality Reduction and Large Scale Optimization" (Now Publishers, 2016 and 2017), and more recently a research monograph on Data Analytics on Graphs (Now Publishers, 2021). Dr Mandic is a 2019 recipient of the Dennis Gabor Award for "Outstanding Achievements in Neural Engineering", given by the International Neural Networks Society. He was a 2018 winner of the Best Paper Award in IEEE Signal Processing Magazine, and a 2021 winner of the Outstanding Paper Award in the International Conference on Acoustics, Speech and Signal Processing (ICASSP). Dr Mandic served in various roles in the ICASSP, Word Congress on Computational Intelligence (WCCI) and International Joint Conference on Neural Networks (IJCNN) series of conferences. He has given about 70 Keynote and Tutorial lectures in international conferences and was appointed by the World University Service (WUS), as a Visiting Lecturer within the Brain Gain Program (BGP), in 2015. Dr Mandic is a 2014 recipient of President Award for Excellence in Postgraduate Supervision at Imperial College and holds six patents.

Convolutional Neural Networks Demystified: A Matched Filtering Perspective Based Tutorial

Ljubiša Stanković, *Fellow, IEEE* and Danilo Mandic, *Fellow, IEEE*

SUPPLEMENTARY MATERIAL

The following material supports the main body of the manuscript and provides further evaluation, quantification, and more details on the various steps and procedures considered.

Notation convention. For convenience of cross-referencing, the equations and figures from the main text body of this article will be denoted with the prefix 'P', for example, (P-1) refers to equation (1) in the article and Fig. P-1 refers to Fig. 1 in the article.

I. Motivation: Mitigating the Curse of Dimensionality

To put the effects of Curse of Dimensionality into perspective, even a modest resolution VGA image, with 640×480 pixels, contains a total of $307,200$ pixels, while a $1,920 \times 1,080$-pixel HDTV image comprises $2,073,600$ pixels. In the context of neural network (NN) based processing of such images, the dimension of the input layer typically equals the number of image pixels so that even for e.g. a low-resolution VGA image which is fed into a relatively standard $1,000$-neuron fully connected hidden layer, the number of NN parameters becomes $307,200 \times 1,000 \approx 3 \times 10^8$ [1], [2], [3]. A subsequent fully connected layer with e.g. $1,000$ neurons would require $1,000 \times 1,000 = 10^6$ additional parameters, so that the computational burden for deep neural networks (DNN), which may involve dozens and hundreds of hidden layers, quickly becomes unmanageable on standard computers.

Advances in computer power have empowered data analysts with the ability to approach DNN learning in a heuristic and ad-hoc manner – a brute force black box approach. On the other hand, through the use of domain knowledge and local information from a system engineering viewpoint, we could both simplify the processing chain and make DNNs more physically interpretable and theoretically better grounded – a subject of this tutorial.

Convolutional neural networks have become a standard tool in data analytics, however, their benefits would be greatly enhanced if they were developed and adopted by an extended community which includes practitioners of systems science who routinely analyse real-world sensor data. To this end, an effort to demystify the operation of DNNs is critical to resolving the issues which arise from their black box nature, while employing any available "domain knowledge" in the form of, for example, some well-understood data association principles would help understand their operation [4], [5] and aid their interpretability and explainability. In this sense, the key overarching question remains that of justifying the use of convolution as an appropriate operator for the detection of features in input data – a subject of this work.

II. Forward path in CNNs

We first provide a brief intuition on the application of the matched filtering perspective of CNNs to image inputs.

A. Intuition for operation on images

Some of the Remarks in the main body of this article can be readily extended to cater for image inputs, as follows.

- Regarding Remark P-1, the same principle holds for images, whereby the presence of a two-dimensional feature, $w(m,n)$, in an image is established through a two-dimensional matched filter which performs the convolution with the "template" $w(-m,-n)$ by sliding this pattern along the image in both spatial directions and taking a dot product with the corresponding image portion.
- Regarding Remark P-5, if an image is considered, then the output image of the convolution filter is of the size $(N - M + 1) \times (N - M + 1)$. There are K such images in the convolutional layer so that the total number of convolution filter weights becomes $K \times M^2$, which is again typically much smaller than the $K \times N^2$ connections in the standard fully connected layer.

L. Stanković is with the University of Montenegro, Podgorica, Montenegro. D. P. Mandic is with Imperial College London, London, United Kingdom. Contact e-mail: ljubisa@ucg.ac.me

- Regarding Remark P-6, for an image, the total number of weights is $K(M^2+1)$.
- Regarding Remark P-9, for images we may use different lengths of 1×1 channel-wise filters to reduce, or even increase (if zero-padding is present) the number of weights in the second step of the "convolutions with 1×1 filters".

B. Relation between convolutional neural networks and standard neural networks

Fig. P-4 provides a simple and intuitive connection between CNNs and standard fully connected NNs. This is further elaborated in Example 1.

Example 1. ***Relation to standard neural networks.*** *To show that the input-output relation in a convolutional layer of a CNN simplifies into that of a standard fully connected neural network (FCNN) as a special case, consider a general element-wise form of the "net input" (before the activation function) at an unindexed neuron, given by*

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

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

When it comes to CNNs, N samples of the signal, $x(n)$, represent the input layer (one training datum). These samples are used to produce K outputs of the first convolutional layer, so that (with a slight abuse in indexing) its input-output relation can be expressed in an FCNN-form as

$$y_k=\sum_{m=0}^{N-1} w_k^1(m)x(m)=w_k^1(0)x(0)+w_k^1(1)x(1)+\cdots+w_k^1(N-1)x(N-1). \tag{2}$$

A comparison between the output of an FCNN layer in (1) and its CNN–like counterpart in (2) with the expression for the convolution filter in (P-8) shows that for $M=N$ the ***standard neural network is a special case of the CNN****. Indeed, for $M=N$ the relation in (P-8), becomes*

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

which is conformal with (2). Since $x(n)$ is defined for $n=0,1,\ldots,N-1$, when no zero-padding is applied to the input signal, this expression can be calculated only for $n=0$ to yield

$$y_k^1=y_k^1(0)=w_k^1(0)x(0)+w_k^1(1)x(1)+w_k^1(2)x(2)+\cdots+w_k^1(N-1)x(N-1),\ \text{for } k=1,2,\ldots,K, \tag{3}$$

which is identical to (2). This is further elaborated in Fig. P-4 which illustrates the operation of the first convolutional layer in CNNs for a range of convolution kernel widths, M, and also supports the choice $M\ll N$.

Note: All results which are derived next for the convolutional layer also hold for the standard, fully connected layer, as a special case with $y_k^1=y_k^1(n)$, $n=0$, and $M=N$.

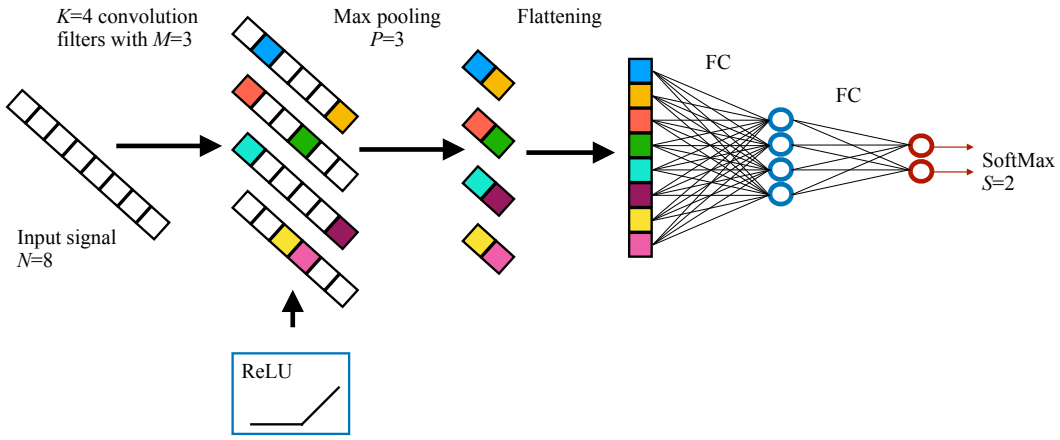

Fig. 1. Information flow in a CNN which consists of: (i) input layer with $N=8$ samples, (ii) convolutional layer with $K=4$ convolution filters (channels) of length $M=3$ to yield $N-M+1=6$ samples at their outputs, (iii) ReLU activation stage, (iv) max-pooling stage with a factor of $P=3$, which reduces the outputs from the convolutional channels to 2 samples each, (v) flattening stage with 8 samples arising from the 4 concatenated outputs of the pooling stage, and (vi) two fully connected output layers.

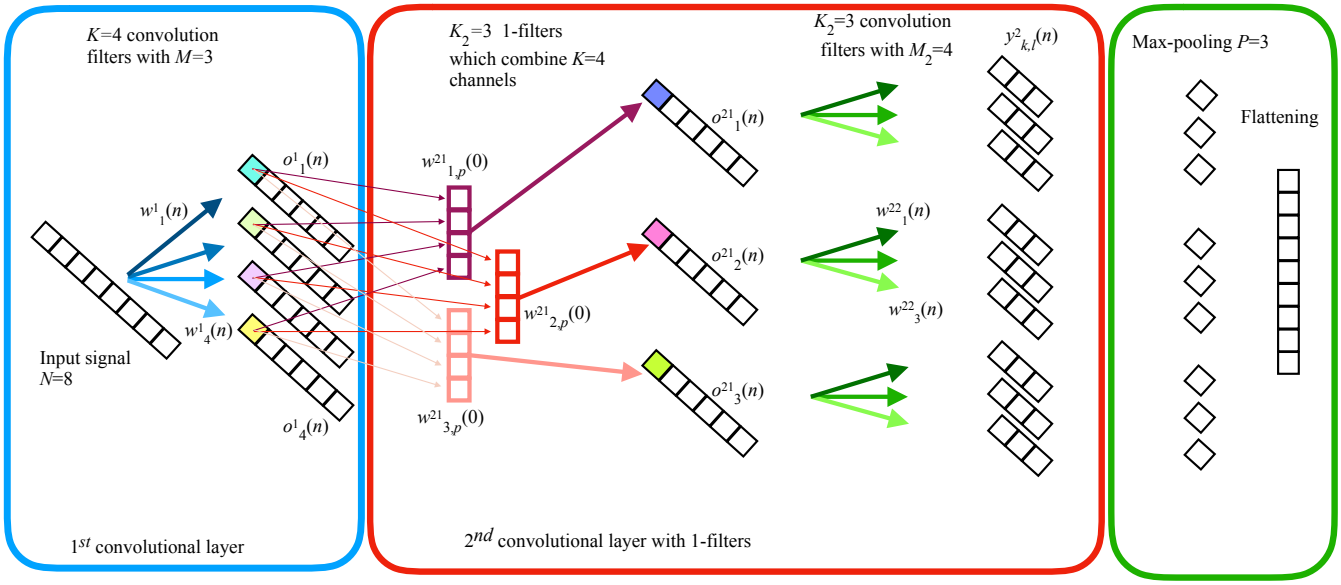

Fig. 2. Illustration of the operation of the convolutions with 1-filters in the 2^{nd} convolutional layer. **Step 1:** The outputs from $K=4$ channels of the first convolutional layer undergo a weighted average along the channel dimension using four-sample weights $w^{21}_{k,p}(0)$, $k=1,2,3,4$. Since there are $K_2=3$ channels in the 2^{nd} convolutional layer, the weighted average is performed $K_2=3$ times based on three independent averaging filters. The total number of the weights in this step is $KK_2=4\times 3=12$. **Step 2:** The so obtained $K_2=3$ outputs are used as inputs to $K_2=3$ channels of a convolution filter, $w^{22}_k(m)$, of length $M_2=4$. The total number of weights in this step is $K_2M_2=3\times 4=12$. The total number of adaptive weights in the two steps is $KK_2+K_2M_2=24$. If the convolution filters were applied directly to the $K=4$ outputs of the first convolutional layer, we would have $K_2=3$ filters for each of $K=4$ outputs (inputs to the second layer) and the length of each filter would have been $M_2=4$, a total $KK_2M_2=48$ weights for the adaptation.

III. Updating Convolution Weights: Back-propagation

The parameters (weights) of a CNN are typically updated in a supervised manner, through a gradient-based learning process known as the back-propagation (BP) algorithm. Given the multi-layer structure of CNNs and since the estimation error can be observed only at the output neurons, the BP algorithm needs to calculate the gradients of the objective function of the optimisation process (e.g. error power) with respect to every single parameter in both the output layer and hidden layers of neurons. These gradients effectively represent the sensitivities of the objective function (e.g. output error power) to each of the network parameters (that is, the proportion of the total error which can be attributed to a given network parameter). Those parameter sensitivities are then used to iteratively update all CNN parameters until a certain stopping criterion is met or the training data set is exhausted.

1) Initialization: Unlike standard linear adaptive systems where the initial weight values are usually set to zero, the initial values of weights in neural networks are typically assumed as random (and different) for every parameter in each channel and layer. Intuition tells us that since the weights $w_k(m)$ multiply with, in general, N_{in} input signal values (at the considered input neurons of any layer), the only requirement is that the choice of the initial weights should preserve the expected energy of the output of the considered layer. 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, on the average, the ReLU activation function will remove negative output values, which account for a half of the expected signal values and thus for half of the signal energy.

Another possibility is to use *uniformly* distributed initial weights, $w_k(m)$, whereby the sum of N_{in} initial weights, $\sum_{m=1}^{N_{in}} w_k(m)$, has a unit variance. Such uniformly distributed weights are defined on the interval

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

Now, because the variance of a uniform random variable is $\mathrm{Var}\{w_k(m)\}=\frac{6}{N_{in}}\frac{1}{3}$, then the variance of a sum of N_{in} such random variables, upon dividing by 2 to account for the effects of the ReLU activation, will be of unit variance. Such initial values are called the *He initial values*.

If the number of output neurons for the considered layer, N_{out}, is also taken into account within weight initialization, then

we arrive at the *Xavier initialization of the weights*, given by [6]

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

The He initial values, when the the output neurons are included, are uniformly distributed within the interval

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

2) Back-propagation in a two-layer CNN: For clarity, consider first the weight update in the simplest CNN which consists of a convolutional layer and a fully connected output layer, such as the network depicted in Fig. P-1.

Convolutional layer. We have seen in (P-8) and Fig. P-5 that for the input $\mathbf{x} = [x(0),\ x(1),\ldots,x(N-1)]^T$, the output signal of every channel (convolution filter), k, of the convolutional layer of the CNN with K filters of the width M, is given by

$$y_k^1(n) = w_k^1(0)x(n) + w_k^1(1)x(n+1) + \cdots + w_k^1(M-1)x(n+M-1) = \sum_{m=0}^{M-1} w_k^1(m)x(n+m), \tag{4}$$

where the superscript $(\cdot)^1$ designates that the variable in hand belongs to the first (convolutional) layer of a CNN. The overall output for every channel of the convolutional layer is then obtained after the bias term is included and upon the application of the ReLU activation function, to yield

$$o_k^1(n) = f\big(y_k^1(n) + b_k^1\big). \tag{5}$$

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

The K channel outputs from the convolutional layer, each of length $N-M+1$, are next stacked (flattened) into a vector of length $K(N-M+1)$, which serves as input to the second, fully connected layer with S outputs, of the considered CNN. Each of the $K(N-M+1)$ so flattened outputs of the convolutional layer, which contains the signal samples

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

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

$$\begin{aligned} y_k^2 &= w_k^2(0)o_1^1(0) + w_k^2(1)o_1^1(1) + \cdots + w_k^2(N-M)o_1^1(N-M) \\ &\quad + w_k^2(N-M+1)o_2^1(0) + \cdots + w_k^2(2(N-M+1)-1)o_2^1(N-M) \\ &\ \ \vdots \\ &\quad + w_k^2((K-1)(N-M+1))o_K^1(0) + \cdots + w_k^2(K(N-M+1)-1)o_K^1(N-M), \end{aligned} \tag{6}$$

for $k = 1,2,\ldots,S$ where the superscript $(\cdot)^2$ designates that the variable in hand belongs to the second (fully connected) layer of a CNN. Note that the number of weights in the k-th FC layer is $SK(N-M+1)$. The index k is used here (as in all other layers) to denote the index of the channel of the output. According to Fig. P-4 (middle row-right), in the case of a FC layer, each output has one sample, which means that the index k in the FC layer (including the output layer) denotes the index of the output signal (neuron).

A commonly used *loss function* (also called the objective function or the cost function) in the assessment of the performance of neural networks is the mean square error (MSE) between the label predicted by the network, y_k^2, and the true label, t_k, which is given by

$$\mathcal{L} = \frac{1}{2}\sum_{k=1}^{S}(y_k^2 - t_k)^2, \tag{7}$$

The target output, t_k, is also called the *desired output or the teaching signal*, and y_k^2 is the value at the k-th output neuron of the FC layer.

Training process. To establish the relations for a gradient descent type of update of the weights in both the convolutional layer and the fully connected layer within a CNN (that is to minimise the convex loss in (7) in an iterative manner through a training process), we shall first consider the convolutional layer, for which the weights of the convolution filters are given in (4)-(5). Then, the gradient weight update is in the form

$$w_k^1(m)_{new} = w_k^1(m)_{old} - \alpha\frac{\partial\mathcal{L}}{\partial w_k^1(m)}\ {}_{|w_k^1(m)=w_k^1(m)_{old}}. \tag{8}$$

The element-wise values of the above gradient of the loss function with respect to the CNN weights are then calculated as

$$\frac{\partial \mathcal{L}}{\partial w_k^1(m)} = \sum_n \frac{\partial \mathcal{L}}{\partial y_k^1(n)} \frac{\partial y_k^1(n)}{\partial w_k^1(m)} = \sum_n \frac{\partial \mathcal{L}}{\partial y_k^1(n)} x(n+m) = \frac{\partial \mathcal{L}}{\partial y_k^1(m)} *_c x(m), \tag{9}$$

where (4) is used for the calculation[1] of $\partial y_k^1(n)/\partial w_k^1(m)$.

Remark 1: Observe that the above gradient of the loss function combines the sensitivities $\frac{\partial \mathcal{L}}{\partial y_k^1(n)}$ and the inputs, $x(m)$, in a way which is precisely the matched filter relation in (P-3).

The sensitivities, $\partial \mathcal{L}/\partial y_k^1(m)$, are also called the *delta error* function, $\partial \mathcal{L}/\partial y_k^1(m) = \Delta_k^1(m)$, and can be evaluated as

$$\begin{aligned}\Delta_k^1(m) = \frac{\partial \mathcal{L}}{\partial y_k^1(m)} &= \sum_p \frac{\partial \mathcal{L}}{\partial y_p^2} \frac{\partial y_p^2}{\partial y_k^1(m)} = \sum_p \frac{\partial \mathcal{L}}{\partial y_p^2} \frac{\partial y_p^2}{\partial o_k^1(m)} \frac{\partial o_k^1(m)}{\partial y_k^1(m)} \\ &= \sum_p \Delta_p^2 w_p^2\big((k-1)M+m\big) u\big(y_k^1(m)\big)\end{aligned} \tag{10}$$

where $u(\cdot)$ is the unit step function which arises as the derivative of the ReLU activation in the calculation of $\frac{\partial o_k^1(m)}{\partial y_k^1(m)}$, while the relation in (6) 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, ouput stage. The error is equal to the difference of the output signal, y_p^2, and the target t_p (which is known during the training process).

In other words, the relation in (10) back-propagates the error from the FC layer 2, denoted by Δ_p^2, to layer 1 and represents a portion of the overall error (observed at the outputs, S, of the FC layer) attributed to a neuron k of the convolutional layer 1, that is, $\Delta_k^1(m)$. In this way, we can calculate all elements, $\partial \mathcal{L}/\partial y_k^1(m) = \Delta_k^1(m)$, of the gradient in the weight update in (8).

The bias terms are updated in the same way as the weights, that is, based on

$$b_{k,new}^1 = b_{k,old}^1 - \alpha \frac{\partial \mathcal{L}}{\partial b_k^1} \Big|_{b_k^1 = b_{k,old}^1} \tag{11}$$

and

$$\frac{\partial \mathcal{L}}{\partial b_k^1} = \sum_n \frac{\partial \mathcal{L}}{\partial y_k^1(n)} \frac{\partial y_k^1(n)}{\partial b_k^1} = \sum_n \frac{\partial \mathcal{L}}{\partial y_k^1(n)} = \sum_n \Delta_k^1(n) \tag{12}$$

Weight updates under max-pooling. If the max-pooling operation is employed at the first convolutional layer of a CNN, then the output $y_k^1(n)$ is produced only for some $n \in \mathbb{M}_k$, where $\mathbb{M}_k$ is a set of indices of nonzero values in the matrix defined by (P-10). Then, the gradient update is adjusted accordingly, to yield

$$\frac{\partial \mathcal{L}}{\partial w_k^1(m)} = \sum_{n \in \mathbb{M}_k} \frac{\partial \mathcal{L}}{\partial y_k^1(n)} \frac{\partial y_k^1(n)}{\partial w_k^1(m)} = \sum_{n \in \mathbb{M}_k} \frac{\partial \mathcal{L}}{\partial y_k^1(n)} x(n+m). \tag{13}$$

Notice that with the max-pooling in place, 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 calculated at $y_k(0)$, $y_k(2)$, …, $y_k(N-2)$. The indicator matrices for the max-pooling, activation, and stride operations are discussed respectively in Examples P-5, P-3, and P-4.

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

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

Without max-pooling, the indices n in $o_F^1(n)$ range from 0 to $K(N-M+1)-1$. Notice that the relation which connects the

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

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

outputs of the convolutional layer and the outputs of the FC layer in (6) can be equally written as

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

It is now obvious that the update of the weights in the fully connected layer, $w_k^2(n)$, can be performed in the same way as that for the convolutional layer, based on

$$w_k^2(m)_{new} = w_k^2(m)_{old} - \alpha \frac{\partial \mathcal{L}}{\partial w_k^2(m)} \Big|_{w_k^2(m)=w_k^2(m)_{old}}, \tag{14}$$

whereby the gradient elements have the form

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

Observe that this relation is *a special case* of (13), for the CNN with $y_k^2(n) = y_k^2(0) = y_k^2$, that is, when the summation over n in (13) reduces to one term only for $n = 0$, that is, when a convolutional layer reduces to a fully connected layer, as elaborated in Example 1.

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 $\partial \mathcal{L} / \partial w_k^2(m)$, which is a constant of unity in the case of the ReLU activation.

3) SoftMax output layer: Some applications require that the output layer gives the probabilities for the decision when classifying the analyzed data. In other words, the output should represent the probabilities of presence of a certain class (for example, dog, cat, bird) in the input, whereby the label that receives the highest probability represents the overall classification decision. Since the output signals of a CNN are mapped to the range $[0,1]$, the values of the target signal also need to be modified to match the output range. Therefore, in the training process, the target (teaching signal) for an output y_k assumes the value $t_k = 1$ when a correct class is detected (as during training we know the signal/image that is being analyzed by a CNN) and $t_k = 0$ for the incorrect decision by a given output neuron.

Since the output, y_k^L, from the last fully connected L-th layer (overall output), may assume various positive or negative real values, the output y_k^L needs to be mapped onto probability-like range of values. This can be achieved using a function of the form

$$P_k = \frac{e^{y_k^L}}{\sum_{i=1}^S e^{y_i^L}}. \tag{15}$$

called the SoftMax nonlinearity, whereby $0 \le P_k \le 1$ and $\sum_{k=1}^S P_k = 1$.

With SoftMax as the output mapping, the loss function needs to be modified accordingly, from the mean square error to the *cross-entropy* form, given by

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

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

The delta error function for the cross-entropy loss function in the output layer can be straightforwardly shown to be 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 (15) 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 desired, there is no weight update if the correct decision, $t_k = P_k$, is produced by the CNN, while all the previous (and the following) relations regarding the back-propagation also hold for the cross-entropy loss.

4) Back–Propagation in a Multi–Layer CNN: We shall now generalize the above illustration of the operation of the back-propagation algorithm over a simple two-layer network to a general multi-layer CNN. Using the same indexing scheme as above, the output of the layer l, $l = 1,2,\ldots,L$, of a general CNN without max-pooling, is defined as

$$\mathbf{y}_k^l = \sum_p \mathbf{o}_p^{l-1} *_c \mathbf{w}_{k,p}^l + b_k^l, \tag{16}$$

where $\mathbf{o}^{l-1}$ is the output of the layer $(l-1)$, as shown in Fig. P-6. The element-wise form of this output is given by

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

Notice that, with this notation, the overall input to CNN (layer 0) is 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 a nonlinear activation function (commonly ReLU in CNNs).

The derivatives in (17) that will be used in the update of neural network weights are calculated as follows

$$\frac{\partial y_k^l(n)}{\partial w_{k,p}^l(m)} = o_p^{(l-1)}(n+m)$$

$$\frac{\partial y_k^{l+1}(n-\mu)}{\partial o_q^l(n)} = \frac{\partial}{\partial o_q^l(n)}\Big(\sum_{p=1}^{K}\sum_{m=0}^{M-1}\Big(o_p^l(n-\mu+m)w_{k,p}^{l+1}(m)\Big) + b_k^{l+1}\Big) = w_{k,q}^{l+1}(\mu)$$

$$\frac{\partial o_p^l(n)}{\partial y_k^l(n)} = f'(y_k^l(n)) = u(y_k^l(n)),$$

where $u(\cdot)$ is the unit step function, that is, the derivative of the ReLU activation.

Gradients of weight update. The weights in a CNN are updated according to the gradient descent direction of the loss function, $\mathcal{L}$, that is, based on

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

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

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

  Remark 2: Observe that, similarly to the case of MSE type of cost function addressed in Remark 1, the backpropagation information flow for the cross-entropy cost function also assumes the form of a matched filter.

- For the standard, fully connected layer, according to (3), 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: Recall that in the CNN parlance, the derivative $\partial\mathcal{L}/\partial y_k^l(m) = \Delta_k^l(m)$ is called the *delta error*. The parameters in an arbitrary (hidden) layer l, should be therefore 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

$$\Delta_k^l(n) = \frac{\partial \mathcal{L}}{\partial y_k^l(n)} = \sum_m \sum_p \frac{\partial \mathcal{L}}{\partial y_p^{l+1}(n-m)} \frac{\partial y_p^{l+1}(n-m)}{\partial y_k^l(n)}$$

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

This equation relates the delta error at the layer l, given by $\Delta_k^l(n) = \frac{\partial \mathcal{L}}{\partial y_k^l(n)}$ with the delta error of the next layer, $l+1$, denoted by $\frac{\partial \mathcal{L}}{\partial y_p^{l+1}(n)}$. This relation will be next rewritten to form the back-propagation relation.

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

$$\Delta_k^l(n) = u(y_k^l(n)) \sum_p \Big(\sum_m \frac{\partial \mathcal{L}}{\partial y_p^{l+1}(n-m)} w_{p,k}^{l+1}(m)\Big) = u(y_k^l(n)) \sum_p \Big(\Delta_p^{l+1}(n) * w_{k,p}^{l+1}(n)\Big). \tag{19}$$

In order to start the recursion (back-propagation) for the update of all weights within a CNN, we must first calculate the delta error in the last layer, denoted by Δ_k^L, as this is the only directly observable error. We have seen that if the mean square error is used as a cost function, then

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

In the same spirit, for the SoftMax output layer we have

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

so that we can easily calculate $\Delta_k^{L-1}(n)$ using (19) and Δ_k^L, and so on.

The above back-propagation of the delta error has been derived for the convolutional layers. For a fully connected layer, as in standard neural networks, we can obtain the corresponding back-propagation rule by rewriting the above relation for the convolutional layer while omitting the convolution operation, as indicated in Fig. P-4 and Example 1. In this way

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

Bias update. The back-propagation of the bias terms obeys similar rules as for the ordinary weights, 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

$$b_{k,new}^l = b_{k,old}^l - \alpha \frac{\partial \mathcal{L}}{\partial b_k^l} \Big|_{b_k^l = b_{k,old}^l} \tag{20}$$

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 (20).

The entirety of the backpropagation algorithm for the update of the weights in CNNs is numerically illustrated in Example 2 on the next page.

IV. Neuron Dropout

Dropout for regularizing deep neural networks. Neural networks are known to perform universal function approximation [7], however, fully connected deep neural networks tend to quickly *overfit* small-scale training datasets owing to the inappropriately large number of network parameters (weights). In CNNs, this problem of the explosion in parameter dimensionality with the increase in the number of hidden layers is mitigated to an extent through convolutional layers, and the max-pooling or output down-sampling (stride) strategies.

Another way of reducing parameter complexity in fully connected DNNs is by breaking correlations between neuron parameters (also known as co-adaptation) through regularization. This can be achieved by *randomly dropping out neurons and the associated weights* from a given DNN architecture, whereby with every neuron we associate a probability, P, of its removal. The dropout operation is typically performed over many independent realisations and yields smaller-scale and better manageable subnetworks, which owing to their random generation all have different architectures. Then, the training is performed in parallel over these subnetworks, whereby the effect of neuron dropout is that weight updates within a given subnetwork are performed with a different "view" of the overall DNN architecture, compared with other subnetworks. The subnetwork weights trained in this way are then recombined into those of the full DNN by averaging over the number of their random realisations. In this sense, *neuron dropout represents a kind of a sparsification at a given layer* and thus provides more robust training especially for small-scale inputs. When it comes to CNNs, the above described dropout can be used within the FC layers, wheareas the convolutional layers employ a specific form called *spatial dropout* whereby the entire convolutional kernels (features) are dropped, together with the corresponding weights connecting these kernels to the preceding and subsequent layers. Dropout can be interpreted as a process regularizing a NN by artificially corrupting the training process with "system noise" (*cf.* artificially corrupting features in CNNs) in order to stabilize the predictions (test stage), as elaborated in [8].

Example 2. *This example complements Example P-9 by providing, in a step-by-step manner, all the numerical values related to the operation of the two-layer CNN network with one convolutional layer and one fully connected layer, considered in Example P-9. The input signal had $N = 8$ samples, and contained a noisy version of one of the two characteristic patterns (features): (i) a variant of the triangular shape pattern,* $\mathbf{feature}_1 = [-0.5, \ 1, \ -0.5] + \nu(n)$*, for which the target output (class) is* $\mathbf{t} = [1, \ 0]^T$ *or (ii) a variant of a rectangular three-sample* $\mathbf{feature}_2 = [1, \ 1, \ 1] + \nu(n)$*, for which the target output (class) is* $\mathbf{t} = [0, \ 1]^T$*, where* $\nu(n)$ *denotes random uniformly distributed noise whose values lie in the region* $[0, 0.3]$*. These noisy signals were further corrupted by additive white Gaussian noise with the standard deviation of* 0.05*, and then normalized to unit energy, as shown in Fig. P-8(a). Convolution filters of* $M = 3$ *samples were used within* $K = 3$ *channels at the convolutional layer. The SoftMax function was used at the output of the FC layer, with two outputs that correspond to the two patterns in the target signal,* $\mathbf{t}$*, and the corresponding target signals,* $\mathbf{t}$*, as described above. The network was trained using* 200 *random realizations of the input,* $\mathbf{x}$*, which were presented 10 times to the network, that is, the training was performed over 10 epochs of 200 random signal realizations. After the training, the network was tested over* 100 *new random signal realizations, which were not seen by the network during the training stage.*

The detailed elaboration and quantification of all the steps which yield the results given in Fig. P-8 are as follows.

***Forward calculation**: From the input signal to the output of the fully connected layer*

• ***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 only* $feature_2$ *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, which gives*
$\mathbf{w}_1^1 = [-0.07 \quad -0.01 \quad -1.47]^T$,
$\mathbf{w}_2^1 = [\ 0.44 \quad 0.14 \quad -0.30]^T$,
$\mathbf{w}_3^1 = [\ 1.15 \quad -1.01 \quad -1.83]^T$.

• ***Convolutions**:* $\mathbf{y}_k = \mathbf{x} *_c \mathbf{w}_k^1 + b_k, \quad k = 1,2,3$ *with the initial bias values* $b_1 = 0, \ \ b_2 = 0$*, and* $b_3 = 0$*, to yield*
$\mathbf{y}_1 = [\ 0.35 \quad 0.49 \quad -0.65 \quad -0.65 \quad -0.69 \quad 0.48]^T$,
$\mathbf{y}_2 = [-0.05 \quad -0.06 \quad -0.28 \quad -0.21 \quad 0.13 \quad 0.37]^T$,
$\mathbf{y}_3 = [\ 0.48 \quad 0.50 \quad -0.77 \quad -1.66 \quad -0.76 \quad 0.71]^T$.

•***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**: With* $P = 3$*, the max-pooling output* $o_k^1(m) = \max\{F_k(mP), \ldots, F_k(mP + P - 1)\}$*, becomes*

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

From Example P-3 and Example P-5, the indicator matrices of the values selected by ReLU, $\mathbf{M}^{ReLU}$*, and max-pooling,* $\mathbf{M}^{MP}$*, are*

$$\mathbf{M}^{ReLU} = \begin{bmatrix} \mathbf{1} & \mathbf{1} & 0 & 0 & 0 & \mathbf{1} \\ \mathbf{0} & 0 & 0 & 0 & \mathbf{1} & \mathbf{1} \\ \mathbf{1} & \mathbf{1} & 0 & 0 & 0 & \mathbf{1} \end{bmatrix}^T \quad \text{and} \quad \mathbf{M}^{MP} = \begin{bmatrix} 0 & \mathbf{1} & 0 & 0 & 0 & \mathbf{1} \\ \mathbf{1} & 0 & 0 & 0 & 0 & \mathbf{1} \\ 0 & \mathbf{1} & 0 & 0 & 0 & \mathbf{1} \end{bmatrix}^T,$$

and will be used to reposition the gradient update calculated with the downsampled, $\mathbf{o}^1$*, to the proper,* $\mathbf{y}^1$*, positions, thus taking into account the positions of the output which survived zeroing by the ReLU and the max-pooling.*

• ***Flattening**:* $N_F = (N - M + 1)P = 2, \quad o_F^1((k-1)N_F + m) = o_k^1(m), \quad k = 1,2,3, \ \ m = 0,1$*, so that*
$\mathbf{o}_F^1 = [0.49 \quad 0.48 \quad 0.00 \quad 0.37 \quad 0.50 \quad 0.71]^T$.

• ***Weight initialization**: For the FC layer, we chose random* $w_k^2(n) \sim \mathcal{N}(0,1)\sqrt{2/6}$*, to give*

$$\mathbf{w}^2 = \begin{bmatrix} -0.47 & -0.06 & -0.05 & -0.81 & 0.62 & -0.18 \\ 0.02 & 0.12 & -0.15 & -0.07 & -0.87 & -0.53 \end{bmatrix}^T,$$

• ***Output**: From the above, the two outputs of the FC layer,* $y_k^2 = \sum_{n=0}^{5} o_F^1(n) w_k^2(n)$*, have the value*
$\mathbf{y}^2 = (\mathbf{w}^2)^T \mathbf{o}_F^1 = [-0.38 \quad -0.77]^T$.

• ***SoftMax**: With* $S = 2, \ \ P_k = e^{y_k^2}/(e^{y_1^2} + e^{y_2^2}), \ \ k = 1,2$*, we have*
$\mathbf{P} = [0.60 \quad 0.40]^T, \quad \mathbf{\Delta}^2 = \mathbf{P} - \mathbf{t} = [-0.40 \quad 0.40]^T, \quad \Delta_k^2 = P_k - t_k$.

Based on the above set-up, the training stage was performed through back-propagation, with the signal values at the different stages of the process as follows.

***Back-propagation**: Delta error, gradient, weight update*

• ***Gradient** in the FC layer, $g_k^2(m) = \Delta_k^2 o_F^1(m)$, was calculated as*

$$\mathbf{g}^2 = \mathbf{o}_F^1(\mathbf{\Delta}^2)^T = \begin{bmatrix} -0.12 & -0.12 & -0.00 & -0.09 & -0.12 & -0.17 \\ 0.12 & 0.12 & 0.00 & 0.09 & 0.12 & 0.17 \end{bmatrix}^T.$$

• ***Weight update** in the FC layer, $w_k^2(m) \leftarrow w_k^2(m) + 0.1 g_k^2(m)$, then takes the values*

$$\mathbf{w}^2 = \mathbf{w}^2 - 0.1\mathbf{g}^2 = \begin{bmatrix} -0.45 & -0.04 & -0.05 & -0.79 & 0.64 & -0.15 \\ 0.00 & 0.10 & -0.15 & -0.08 & -0.89 & -0.56 \end{bmatrix}^T.$$

• ***Delta error back-propagation** in the convolutional layer, $\Delta_k^1(m) = \sum_p \Delta_p^2 w_p^2(m)$,*
$(\mathbf{\Delta}^2)^T\mathbf{w}^2 = [0.18,\ 0.06,\ \mathbf{-0.04},\ 0.29,\ -0.62,\ -0.16]$, *now follows as*

Repositioned $\{(\mathbf{\Delta}^2)^T\mathbf{w}^2\} = \begin{bmatrix} 0.18 & 0.06 \\ -0.04 & 0.29 \\ -0.62 & -016 \end{bmatrix}$, *following* $\mathbf{o}_F^1 \rightarrow \mathbf{o}^1$.

• ***Repositioning** the elements of $\mathbf{\Delta}_k^1 = (\mathbf{\Delta}^2)^T\mathbf{w}^2$ at the positions defined by $\mathbf{M}_k^{MP} \odot \mathbf{M}_k^{ReLU}$ is based on the indicator matrices for the ReLU and max-pooling operations, as in From Example P-3 and Example P-5, to yield*
$\mathbf{\Delta}_1^1 = [0 \quad 0.18 \quad 0 \quad 0 \quad 0 \quad 0.06]^T$,
$\mathbf{\Delta}_2^1 = [\mathbf{0} \quad 0 \quad 0 \quad 0 \quad 0 \quad 0.29]^T$
$\mathbf{\Delta}_3^1 = [0 \ -0.62 \quad 0 \quad 0 \quad 0 \ -0.16]^T$
where $\odot$ is the Hadamard element-by-element product.

• ***Gradient** in the convolutional layer, $g_k^1(m) = \Delta_k^1(m) *_c x(m)$, now takes values $\mathbf{g}_k^1 = \mathbf{\Delta}_k^1 *_c \mathbf{x}$, given by*
$\mathbf{g}_1^1 = [-0.03 \quad 0.03 \quad 0.06]^T$,
$\mathbf{g}_2^1 = [\ 0.01 \quad 0.20 \quad 0.12]^T$,
$\mathbf{g}_3^1 = [\ 0.06 \quad -0.02 \quad 0.06]^T$.

• ***Weight update** in the convolutional layer $w_k^1(m) \leftarrow w_k^1(m) - 0.1 g_k(m)$, produces the weights*
$\mathbf{w}_1^1 = [-0.06 \quad -0.01 \quad -1.47]^T$,
$\mathbf{w}_1^2 = [\ 0.45 \quad 0.13 \quad -0.31]^T$,
$\mathbf{w}_1^3 = [\ 1.12 \quad -1.01 \quad -1.84]^T$.

• ***Bias update**, $b_k \leftarrow b_k - 0.05\sum_n \Delta_k^1(n)$, follows similarly to the weight update*
$\mathbf{b} \leftarrow \mathbf{b} - 0.05([1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1]\mathbf{\Delta}^1)^T = \mathbf{0} - 0.05[0.24 \quad 0.29 \quad -0.78]^T$.

• ***New iteration** of the backpropagation algorithm starts with the new signal for which the target was, $\mathbf{t} = [0, \ 1]^T$, which is given by*
$\mathbf{x} = [-0.10 \quad -0.12 \quad -0.05 \quad -0.24 \quad 0.89 \quad -0.35 \quad -0.02 \quad -0.00]^T$,
With the new (updated) weights, $\mathbf{w}^1$ and $\mathbf{w}^2$, and bias $\mathbf{b}$, **go back to the first step and continue the recursion**.

After the first training cycle is completed, the process is repeated with a new noisy input signal, $\mathbf{x}$ as in Fig. P-8(a), randomly assuming the presence of either $\mathbf{feature}_1$ or $\mathbf{feature}_2$ at a random position within the signal. The results of the training and testing process are visualized in detail in Fig. P-8.

V. Principle of the matched filtering interpretation of CNNs for images

Based on the above introduced perspective on the matched filtering interpretation of the operation of CNNs for temporal signals, we next provide an intuition for the extension of this approach to CNNs for images. Such an extension is non-trivial, as both the cross-correlation and convolution for images exhibit intricacies which are not present in 1D signals; this is therefore out of scope of the present work. For intuition, we here provide a visualisation which confirms the validity of the matched filtering interpretation of image CNNs. The full analysis is a subject of our ongoing work, with some initial insights available from [9].

Consider a CNN architecture for the recognition of handwritten digits from the MNIST database, given in Figure 3. The CNN aims at classifying the handwritten digit into ten categories, $\{0,1,\ldots,9\}$. Some examples of the handwritten digit "zero" from the MNIST database are given in Figure 4.

To illuminate the generality and potential of the above introduced matched filtering perspective for signals, we next examined the feasibility of the matched filtering interpretation for image-CNNs, akin to that for temporal signals presented above. To this end, we employed 10 convolutional kernels (one per the handwritten digit) of the same sizes as the original MNIST images of handwritten digits, that is 20×20 pixels. Such kernels were randomly initialised and trained by backpropagation. The results in Fig. 5 confirm that not only the trained convolutional kernels adopted the shapes of the corresponding digit classes – precisely the matched filtering operation in the same way shown above for temporal signals – but also this domain knowledge aware architecture achieved high accuracy of 89 % while employing only one convolutional layer and one output layer. Full analysis

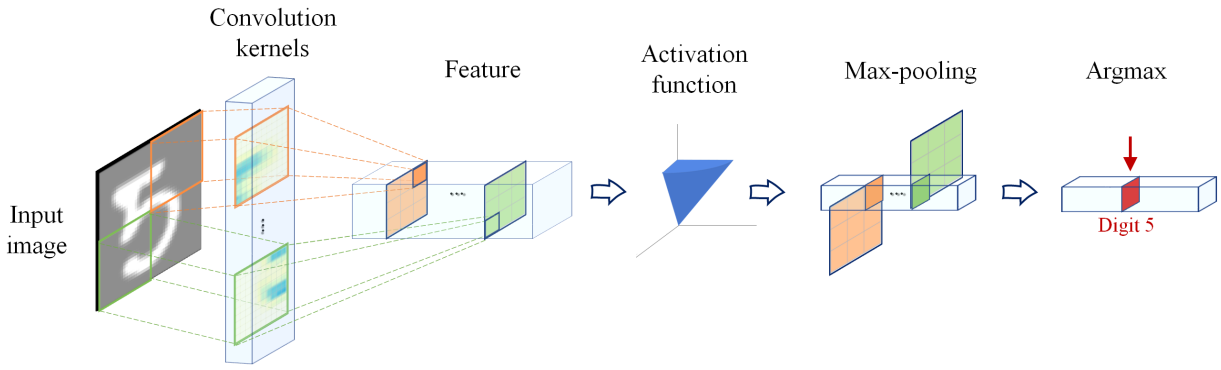

Fig. 3. A single-layer CNN architecture for the recognition of handwriten digits. The images are of size 20×20 pixels.

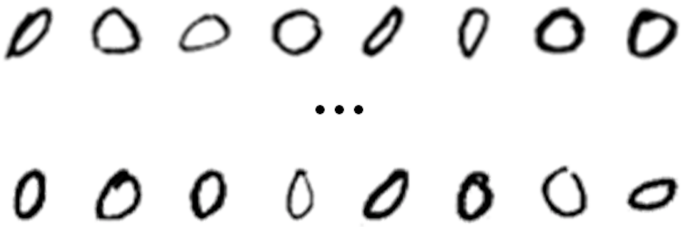

Fig. 4. Exemplar handwritten digits "0" from the MNIST database.

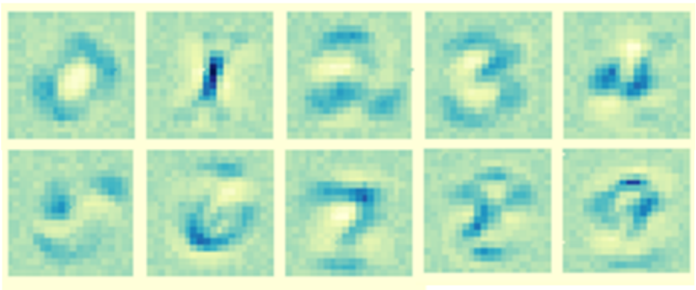

Fig. 5. Convolution kernels for the classification of MNIST handwritten digits, after training from a random initialisation. Observe the validity of the proposed matched filtering perspective, as the learned kernels adopted the shapes of the ten considered digit classes.

and applications of such an approach, together with a matched filtering perspective of multi-task learning, is subject of our ongoing work. Some initial results and intuition along this research direction can be found in [9].

REFERENCES

[1] M. H. Hassoun *et al.*, *Fundamentals of artificial neural networks*. MIT press, 1995.
[2] B. Yegnanarayana, *Artificial neural networks*. PHI Learning Pvt., 2009.
[3] K. Gurney, *An introduction to neural networks*. CRC press, 2018.
[4] Z. Zeng and J. Wang, "Design and analysis of high-capacity associative memories based on a class of discrete-time recurrent neural networks," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 38, no. 6, pp. 1525–1536, 2008.
[5] S. Yang, Z. Guo, and J. Wang, "Robust synchronization of multiple memristive neural networks with uncertain parameters via nonlinear coupling," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 45, no. 7, pp. 1077–1086, 2015.
[6] S. Haykin, *Kalman filtering and neural networks*, vol. 47. John Wiley & Sons, 2004.
[7] D. Mandic and J. Chambers, *Recurrent neural networks for prediction: Learning algorithms, architectures and stability*. Wiley, 2001.
[8] K. Matsuoka, "Noise injection into inputs in back-propagation learning," *IEEE Transactions on Systems, Man ad Cybernetics*, vol. 7, no. 1, pp. 436–440, 1992.
[9] S. Li, X. Zhao, L. Stankovic, and D. Mandic, "Demystifying CNNs for images by matched filters," *arXiv preprint arXiv:4547556*, 2022.